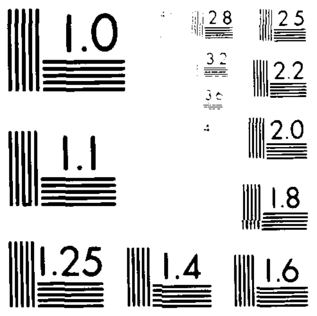
1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

20

Technion-Israel Institute of Technology

Computer Science Department

12

DIMUI - IDMS User Manual

Version 1.2

by

Allen Reiter

Technical Report #101

June 1977

Acknowledgment This work was supported by the U.S. Army Computer Systems Command through funding provided via the Army Research Office Grant Number DAAG29-76-G-0247 and via the Army Research & Standardization Group (Europe) Grant Number DAERO-77-G-006.

4[illegible]573

FIGURES

TABLE OF CONTENTS

1. INTRODUCTION

DIMUI is a general purpose model for database performance evaluation. When supplied with certain parameters and augmentation routines, it becomes a model of a specific generalized database management system. In this document we describe how to use the DIMUI IDMS model.

We include only the minimum of information necessary to model IDMS databases and processes. The reader who is interested in the full DIMUI capability is referred to references [1]-[4]. It is assumed the reader is familiar with reference [5], the IDMS user manual.

DIMUI consists of a set of programs which make up three stages of the main model as well as some utility programs. The first two stages of the model are concerned with translating a user's specification of his application (data description and a specification of each task which he wishes to model) into a very-low-level internal form (*instruction streams* for a virtual data machine VDM; one stream for each task). This internal form is still largely machine-configuration-independent. The last stage of the model accepts a description of the configuration and of the job mix (i.e. the different instruction streams which are to execute concurrently) and produces performance statistics.

The model as described here is intended to be used by a data administrator to evaluate the performance of different possible file designs. He can, with relative ease, evaluate in the context of certain processing tasks the effects of using *prior* and/or *owner* pointers in set representation, or of changing the *location mode* of certain records, or of changing the page size. He can also experiment with the computer configuration, in particular with changing the file extents in secondary storage and with changing the types of storage devices and how they are connected to the main memory. Other useful parameters which he can vary are the number of buffers available at run-time, and the amount of journalizing activity (writing *before* and *after* records). Finally, he can evaluate the system response in an interactive environment by varying the job mix parameters and measuring the amount of mutual interference by concurrent tasks. Not described here is the use of DIMUI by a system designer (e.g. in redesigning parts of the IDMS system); see [1].

The first stage of the model is a preprocessor which translates data descriptions which resemble IDMS DDL and task descriptions which resemble IDMS DML into standard-form internal tables and subroutine calls (this is the so-called SIDBL language). Stage 1 thus forms the interface between the IDMS user and the rest of the model.

The second stage of the model is the Data Modeller (DM). The SIDBL data and task descriptions which are its input are concerned only with *logical* constructs; DM translates these into *physical* page-oriented instruction streams. This translation takes into account such parameters as page size, location mode of the records, number of pages in each area, etc. These instruction streams, however, are not immediately *executed*, but form the input into the final stage of the model. It may be useful to think of DM as a model of the IDMS run-time data manipulation routines; the latter do not perform the I/O operations themselves, but make calls on the operating system services as necessary. DM writes these calls as instruction streams, instead of executing them. (Some functions such as buffer management, which are part of the IDMS run-time routines, are however also postponed until the final stage of the model.)

Stage three is the Execution Modeller (EXM). It is a very detailed model of a multi-programming operating system and of the relevant computer hardware (processors, main memory, channels, disks, and tape units). It contains models of a job scheduler, of resource handlers for each of the hardware resources, and of the hardware resources themselves. Each I/O action is modelled in great detail, and the computation of the completion time for each action takes into account such dynamically-changing parameters as the current disk-arm location, current rotational position, etc. It computes the total running time of the job mix, the system throughout, and various utilization statistics which may be useful in pinpointing the system bottlenecks.

2. INPUTS TO THE MODEL

The user must supply descriptions corresponding to DDL, DML, and DMCL statements in IDMS, as well as some control information for the model. These are briefly described in this section, and in more detail below.

2.1 Data Description

A. Area description cards; one for each area. Specifies the page size for the area, which buffer pool is used for the pages, whether or not *before* and/or *after* images are being written to a journal tape, and the identity of the journal tape unit.

B. Record description cards; one for each record type. Specifies the record name, its size, the number of records of this type in the data base, the area in which the records are stored, and the location mode of the records. (CALC or via a specified set.)

C. Set description cards; one for each set type. Specifies the owner record name, the member record name(s), the number of member occurrences per set occurrence (usually specified as a probability distribution), several parameters connected with the relative placement of the members in storage (explained later), and the set organization parameters (i.e. whether owner and/or prior pointers are utilized)

2.2 Task Description

A task description is a subprogram written in a procedural language. This language consists of DML-like commands, plus control statements (for iteration, conditional execution, and the like) in a host language. The current version of DIMUI-IDMS uses FORTRAN as the host language. A library of up to 15 different tasks is supported. Each task is compiled by the Fortran compiler, a link-edit is performed on the Stage 2 programs, and the desired task is selected when running Stage 2 by a BEGTSK card (see Section 10.1.2.15). Under development (to be included in an expanded

Stage 1 capability) is an interpreter for a special language (not described here) which would obviate the need for compiling and link-editing whenever a task description needs to be modified.

2.3 Media Description

A. Extent descriptions. For each area there are one or more extent description cards, corresponding to each contiguous storage area on one disk. An extent descriptor identifies the disk unit, the starting cylinder, the number of logical tracks per cylinder (an extent may occupy fewer tracks than fit onto a physical cylinder), the number of pages per track, the number of pages in this extent, and the amount of space each page takes on the disk (including overhead for keys, gaps, etc.).

B. Buffer pool descriptions. Each area has one pool of core-storage buffers available to it during execution. One such pool may serve one or more areas. A buffer pool description card (one for each pool) specified the number of buffers in the pool.

C. Channel descriptions. In our model system architecture, each storage device (disk or tape unit) is connected to the main memory via one or more channels. A channel descriptor specifies which units are connected to it, and the servicing priority among these units.

D. Unit descriptions. Each storage unit (disk or tape) has one descriptor associated with it. This descriptor specifies the unit type

E Processor description. Each central or peripheral processor in the system has one such descriptor associated with it; this specifies the speed of the processqr and the type of processing of which it is capable. (Current implementations of IDMS run on single-processor systems; thus only one such card needs to be supplied, which specifies only the CPU speed.)

2.4 Job Load Description

After a library of tasks (as instructions streams) has been produced by the Stage 2 modeller, EXM must be told *which* of these tasks are to be executed, and *when*. If only one task is being timed, the job load description merely specifies which task is to be run. If a multiprogrammed multi-job environment is being simulated, the job load description specifies the degree of multiprogramming, and the sequence of tasks (possibly with interspersed delays, simulating user "think time" for an interactive environment) to be executed, together with their execution priorities. The simulation may terminate after a prespecified time, or after the completion of a specified task, or after the entire job load has been run. The first mode is useful for measuring the instantaneous throughput of a heavily-loaded system; the second, the interference of the "other" processing to the run-time of a specified task; the third, for measuring the total elapsed time required to finish a batch of jobs.

2.5 Control Information

The user specifies which stage of the model is being run, the seed for the random-number generator, which system dumps are desired (normally, these are of no interest to the user, hence none should be requested), and some task identification information transferred from Stage 2 to Stage 3

2.6 Specifying Input Values Probabilistically

Most values which need to be specified by the user (e.g. number of records per set) may be specified as probability distributions. Two types of distributions are currently supported: *discrete*, where the user lists the admissible values together with their respective probabilities, and *continuous*, where the user indicates the minimum and maximum of the range and the expected value. (A third type of distribution, describing the probability of records with different keys hashing to the same page address, is generated internally by the Stage 1 Modeller.)

3. OUTPUTS OF THE MODEL

The output of the final stage of the model is the total (simulated) elapsed time, given as the *master clock* reading (MSTCLK). In addition, the following other information is currently produced:

1) Processor utilization: how much of the time each of the processors was active;
2) Channel utilization: how much of the time each one of the channels was active;
3) Device utilization: how much of the time each one of the units was active, and how much of the time it spent waiting for a channel to become available in order to initiate an operation which had been requested; also (for disks) the number of I/O requests which found the disk arm already in place, so that no "seek" was required;
4) I/O request summary: how many I/O operations were initiated on behalf of each task or series of tasks.

Finally, the system "throughput" is computed. In order for this number to be meaningful, the user must include in his process description for each task a command asking the model to count weighted "work units" at various places throughout the task. For example, processing a transaction of a certain type can count for x number of work units, while processing each record in a different program can count for y units. (The user must assign x and y.) System throughput is defined as the number of work units processed per second. The total number of work units for each task or series of tasks is also computed.

4. AN EXAMPLE OF AN IDMS DATA BASE

The following example is adapted from [5]. A customer-order system consists of a file of CUSTOMERs and of PRODUCTs. We must keep track of CUSTOMER-ORDERs, each one of which consists of a variable number of ORDER-ITEMs. The IDMS structure (Bachmann) diagram for the files is shown in Figure 1. We will make use of this application in subsequent examples.

For the purpose of the model it is convenient to show in diagrammatic form also the location mode of each record. For this purpose, we modify the Bachmann diagram as follows. If a member record is stored VIA the set, the set is shown as a solid line from the owner to the member; if the location mode of the member records is unrelated to the set occurrence, then the set is shown as a dashed line. We also attempt to show the solid-line connection as *trees*: thus, if record B is located via a set whose owner is A, we will try to place B immediately under A. Thus, ignoring dashed-line connections, the schema looks like a forest of trees, whose root nodes are all of the records whose LOCATION MODE IS CALC.

Assuming that the CUSTOMER and PRODUCT records are CALC, that the CUSTOMER-ORDER record is located via the ORDER set, and that the ORDER-ITEM record is located via the ITEM set, our structure diagram for the example is shown in Figure 2. Note that we also indicate the AREAs in which the records are located.

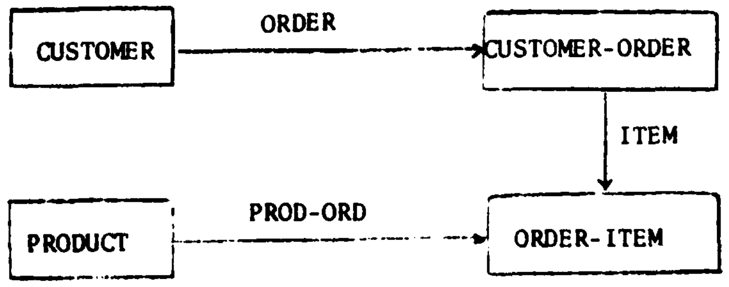

Figure 1. IDMS structure (Bachmann) diagram

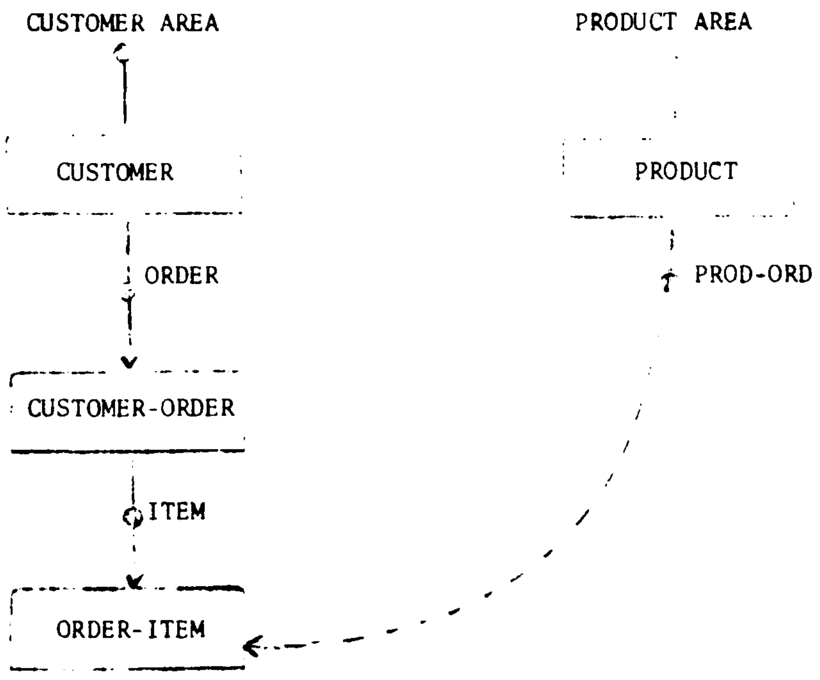

Figure 2. The model schema for the data structure of Figure 1, showing CUSTOMER and PRODUCT records as CALC, the CUSTOMER-ORDER record located VIA the ORDER set, and the ORDER-ITEM record located VIA the ITEM set.

Note: In the example, long element (record and set) names are used for readability. The current system however restricts all element names to being *four* characters long.

5. HOW TO WRITE A TASK DESCRIPTION

A task description is a subroutine written in Fortran IV. Up to 15 different task descriptions can be incorporated into the model at the same time. Each task description (whose name must be TASKn, $6 \leq n \leq 20$) is compiled as an ordinary Fortran external subroutine, and the object module is saved in a system library. Then a link-edit of the entire Stage 2 model must be performed, to obtain a load module of Stage 2. When Stage 2 is run, the desired task is specified by indicating n (of TASKn) on the BEGTSK card.

Please note that a task description in general describes a family of processing tasks. Consider the following task description in English (the equivalent DIMUI description is given later): "For a given customer, print out the information on all of the products which he has ordered". This involves traversing the ORDER set, and (for each CUSTOMER-ORDER) its ITEM set and accessing the PRODUCT record corresponding to the ORDER-ITEM. If the model is run twice (changing the random-number seed) with this task, two different instruction streams are generated, corresponding to performing this task for two different customers. Depending on such parameters as the distributions of the number of members in the ORDER and ITEM sets, the execution time differences between the two instances of the "same" task can be significant indeed. Thus, in general a task model must be run several times to obtain a statistically valid timing estimate. This can be done either by repeating Stage 2 with different starting seeds, or (equivalently) by writing a task description so that it repeats the task the desired number of times (e.g. "For five given customers,...").

The statements in a task description subroutine are almost entirely calls to DIMUI-IDMS interface subroutines. The exceptions are Fortran control statements of the form DO ... for looping, IF ... for conditional execution, and GOTO for branching. Thus very little knowledge of Fortran (but a good knowledge of IDMS) is required to write a task description.

We now give descriptions for each subroutine in the DIMUI-IDMS interface.

5.1 Stels

IDMS uses the concept of *currency* to define implied operands for most DML commands. In DIMUI, however, there are no implied operands; the intended record or set occurrence must be explicitly indicated. (This has both advantages and disadvantages.) This accounts for the major differences in the respective DML's between IDMS and DIMUI-IDMS.

An element (records and sets are collectively called *elements*) occurrence is indicated by a special "data base pointer variable" called a *stel* (*st*ack *el*ement). From the point of view of Fortran, a stel is an ordinary integer variable. From the point of view of DIMUI, there is a great deal of system storage associated with each stel, in which the system keeps track of the simulated data base environment. Some DIMUI commands return a stel; others expect to receive one or more stels as input parameters, which may be changed ("moved" to other places in the data base). A stel is associated with each element in use by the task. When the task has no further use of the data element, the corresponding stel should be released (STEND command).

Occasionally, several stels may be pointing at the same element occurrence. This is usually because one of these stels will be moved to another element occurrence (e.g. to the next record in a set) while another still points at the original element.

The notion of stels is fundamental to the DIMUI-IDMS language. The reader who has trouble grasping this concept is urged to scan briefly the commands and to study the examples in Section 12.2, and then reread the command descriptions.

5.2 The FIND verbs

There are several distinctly different operations in IDMS associated with the FIND verb. They are modelled by distinct DIMUI commands. In these and other commands, lower-case letters denote stels, while upper case letters denote element names or other parameters. All examples refer to Figures 1 and 2.

5.2.1 FINDST (s,ELNAME,RECNAM,r,VERIFY)

Purpose: Find an occurrence of the indicated element (set or record). "Finding a record" has the same meaning as in IDMS: "Finding a set" means finding the owner of the set and then positioning the stel at the set occurrence (see subsequent commands).

Parameters:

s	- (output) stel for target element occurrence;
ELNAME	- (input) name of the target element (Hollerith);
RECNAM	- (input) 0 for first form; name of bound node (Hollerith) for second form;
r	- (input) 0 for first form; stel which specifies record occurrence (second form);
VERIFY	- (input) .FALSE. if further work will be done with this stel; .TRUE. if no further use of this stel will be made (in this case the s parameter has no meaning on output).

Usage: In the first form, a completely random element occurrence is indicated. Thus CALL FINDST(s1, 'CUSTOMER', 0,0,.FALSE.) would position the stel s1 at a random customer occurrence (and at the same time generate the instruction stream for accessing this record). CALL FINDST(s2, 'ITEM' 0, 0, .FALSE.) would simulate the following sequence of actions: first, a random CUSTOMER record is accessed, then a random occurrence of a CUSTOMER-ORDER record in its ORDER set is accessed. (In the real IDMS, *two* FIND commands would have been required to accomplish this, one for each record type. In DIMUI-IDMS, the FIND for the random CUSTOMER record is performed automatically, since the CUSTOMER-ORDER record is "located" VIA the ORDER set, whose owner is the CUSTOMER record.) Finally, at the end of the command the stel s2 is positioned at the ITEM set, allowing such further operations as FINDNX (see below). The command CALL FINDST(s3, 'ORDER-ITEM', 0,0,.FALSE.) would cause the fetching of *three* records, since both a CUSTOMER and a CUSTOMER-ORDER record need to be fetched in order to locate an ORDER-ITEM record.

In the second form of the FINDST command, the target element is not completely random, but is relative to another given element occurrence. Thus, suppose that stel s1 already points at a specific CUSTOMER record, and that we want to access a random ORDER-ITEM record for *that* customer. This is accomplished by CALL FINDST(s4, 'ORDER-ITEM', 'CUSTOMER', r1, .FALSE.). Note that (because of the structure in Figure 2) a random 'CUSTOMER-ORDER' in *this* CUSTOMER's ORDER set is implicitly fetched.

In the second form, the stel r need not point directly at the given element occurrence, but may point at one of its descendants. Thus, assume that stel r2 is pointing at an occurrence of a CUSTOMER-ORDER record (or, equivalently, at an occurrence of an ITEM set). The command CALL FINDST(s5, 'ORDER-ITEM', 'CUSTOMER', r2, .FALSE.) means "Find a random ORDER-ITEM record in the descendant set of this CUSTOMER", i.e. is equivalent to the situation had r2 been pointing directly at the *owner* of the ORDER set to which the CUSTOMER-ORDER belongs. Specifically, the intended ORDER-ITEM need not be a descendant of the CUSTOMER-ORDER record at which r2 points; the specified *bound* between the input r2 occurrence and the output s5 occurrence is the CUSTOMER element. Compare CALL FINDST(s6, 'ORDER-ITEM','CUSTOMER-ORDER', r2, .FALSE.) with r2 as above; here, a random ORDER-ITEM descendant of *this* CUSTOMER-ORDER is intended, since the specified bound element is CUSTOMER-ORDER.

The VERIFY parameter for either form has the following motivation. Frequently, a task must make a data base access to examine one specific element, but does not use it for further traversals. E.g. in validating a transaction some fields may have to be checked against values stored in the data base. In such cases, the task will not want to reference the stel again, and need not even name it. For example, CALL FINDST(dummy, 'PRODUCT', 0,0,.TRUE.) simulates the fetching of a random PRODUCT record, possibly to check if such a product exists, but intends no further action with this product record.

Error Stops:

26 - ELNAM or RECNAM do not appear in the schema.

5.2.2 FINDDN(r,s,mode)

Purpose: Find an occurrence of a member in the current set occurrence. This member may be *last* in the set, *random* in the set, or a sequence number (first, second, etc.) may be specified.

Parameters:

```
r    - (output) points at the specified record occurrence.
s    - (input)  points at the set occurrence.
mode - (input)  -1 means  "last in set"
                 0 means  "random in set"
               n>0 means  "n'th member in the set".
```

Note: On end-of-data (if the set is empty or has fewer than n members) r is returned as negative. -r in this case points at the last element of the set, and is an active stel.

Usage: s will usually have been positioned at the desired set occurrence by a previous FINDST command. Assume that s1 points at an occurrence of the ORDER set. Then CALL FINDDN(r1,s1,-1) fetches the last CUSTOMER-ORDER record; while CALL FINDDN(r1,s1,0) will fetch a random member of the ORDER set. In our case, the latter can be accomplished equivalently by CALL FINDST(r1,'ORDER-ITEM','ITEM',s1,.FALSE.).

Error Stops:

92 - s is not a valid stel, or there is an error in the "set bounds" descriptor (see 6.4).

5.2.3 FINDNX (r,s)

Purpose: Find next of set (similar to IDMS DML command).

Parameters:

r - (input and output) stel for current and next record occurrence in the set

s - (input) points at the set occurrence.

Note: On end of data (if the input value of r is for the last element in the set), r is returned as negative. -r is the value supplied as input.

Usage: If the set is accessed via its owner, s would have been obtained by a previous FINDST command. In order to traverse the member occurrences, the first member is accessed by CALL FINDDN(r,s,1); this sets up r. Successive commands CALL FINDNX(r,s) then move r through each member in the set. This is typically done in a loop, from which the program exists when r < 0.

If the set is accessed via a member occurrence, s would have been obtained by "establishing" the set (see ESTSET command, 5.6.1). At this point, r points at the member occurrence, whose ordinal number in the set is random The remainder of the set can be traversed in a loop as before. If it is desired to leave a stel pointing at the original record, r can first be "dittoed" to another stel by the DITTO command (see 5.6.2).

Error Stops:

93 - r or s is not a valid stel, or there is an error in the "set bounds" descriptor (see 6.4).

5.2.4 FINDOW(s,r)

Purpose: Find owner of set (similar to IDMS DML command)

Parameters:

s - (input and output) on entry points to the set occurrence
on exit, points to the owner record occurrence.

r - (input) points to the current member of the set.

Usage: If the set has been accessed via its owner, s was set up by a previous FINDST command, and r by a previous FINDDN command (possibly modified later by FINDNX commands).

More commonly, the set would have been accessed via a member (pointed to by r). In this case, s would have been obtained by "establishing" the set (see ESTSET command, 5.6.1). For an example of use, see 5.6.1.

Note: The simulated activity depends on the set organization specified in the data description. If "owner pointers" are specified, the owner is fetched directly. Otherwise, the entire set is traversed (starting with r) until the owner is reached. In either case, r is unchanged.

Error Stops:

See FINDNX, which is called repeatedly if owner pointers are not specified.

5.3 The STORE verbs

There are several different IDMS-DIMUI commands corresponding to the IDMS STORE verb. First, there is a difference whether the record being stored is CALC or is located VIA a certain set. (For the latter, a stel pointing at the current set occurrence has to be supplied — remember that DIMUI does not keep track of currency.) Also, for reasons peculiar to the modelling process, there is an inherent distinction (for records stored VIA a set) between adding one "random" record to an existing set, or "sequentially" building the entire set occurrence one record at a time. There are different commands corresponding to the different cases. In all cases, note that DIMUI does not support the notion of "automatic" set membership; explicit INSSET verbs must be used to add a record to each set of which it is a member (see 5.5.1).

5.3.1 SEARCH(r,RECNAM,.TRUE.)

Purpose: To store a CALC record into the data base. (This implies searching a list of records with the same hash key to check whether the reocord already exists, hence the name SEARCH.)

Parameters:

r	- (output) stel for the newly-created record.
RECNAM	- (input) Name of the record to be stored (Hollerith). The third parameter must be specified as .TRUE. .

Error Stops:

26 - RECNAM does not appear in the schema.

5.3.2 STVIA(s,NEWREC,OLDREC,r)

Purpose: To add one random record (whose location mode is VIA a set) to a data base.

Parameters:

s	- (output)	stel for the newly-created record.
NEWREC	- (input)	Name of the record to be stored (Hollerith).
OLDREC	- (input)	0 for first form; Name of bound element (Hollerith) for second form.
r	- (input)	0 for first form; stel which specifies a record occurrence (second form).

Usage: The usage is analogous to FINDST (see 5.2.1). In the first form, a completely random element occurrence is indicated. For example, to add a new CUSTOMER-ORDER record to the data base (without explicitly specifying the corresponding CUSTOMER record) the command CALL STVIA(s1,'CUSTOMER-ORDER', 0,0) is used. This will cause a random CUSTOMER record to be accessed, and a new CUSTOMER-ORDER record to be stored via its ORDER set.

For the second form, suppose that r already points at a CUSTOMER occurrence, and that we want to store a new CUSTOMER-ORDER record for *this* customer. This can be done by CALL STVIA(s2, 'CUSTOMER-ORDER', 'CUSTOMER',r).

As in FINDST, r need not point directly at the given element occurrence, but may point to one of its descendants. The OLDREC parameter specifies the bound element.

Error Stops:

26 - NEWREC or OLDREC do not appear in the schema.

5.3.3 STORNX(r,s)

Purpose: To store the next record of a set occurrence, all of whose members are being stored by the current task. The location mode of the records must be via this set.

Parameters:

r - (input and output) stel for the current/next record occurrence, which has just been stored;

s - (input) stel for the set occurrence.

Note: On end-of-data (the set has been filled to its allotment), r is returned as negative (-r points at stel for the last record stored via the set).

Usage: In general, s will have been set up by the FINDST command (5.2.1). Before the first record is added, r must be "dittoed" from s (see DITTO command, 5.6.2). This in effect sets r to point to the "0'th occurrence" of the member. Thereafter, STORNX is called repeatedly in a loop until a negative r is returned; whereupon, the stel pointed to by -r should be released (see STEND command, 5.6.3).

5.4 The MODIFY and DELETE verbs.

The MODIFY verb in IDMS DML has an exact counterpart in DIMUI. To model the DELETE verb, note that in general IDMS merely marks the record for deletion; i.e. from the point of view of the model, the effect of a DELETE verb is similar to that of the MODIFY. Hence both are modelled by the same command, MD.

5.4.1 MD(r)

To modify the current record occurrence, or to delete the record occurrence.

Parameter:

r - (input) points at the record occurrence to be modified or deleted.

Notes:

1. Modification of the key of a CALC record is not currently supported.
2. If the modification causes a change in the set order for any set occurrence of which this record is a member (i.e. the data items in the record are control data items for the set order) the user must first DISCONNECT and then CONNECT the record for each set in question (see 5.5). First the set must be "established" (see 5.6.1). Then the record is DISCONNECTed. Then a new predecessor is set up by the FINDDN command (5.2.2). Finally, the record is reCONNECTed to the set.

5.5 The CONNECT (INSERT) and DISCONNECT (REMOVE) verbs.

IDMS-DIMUI does not currently support the notion of AUTOMATIC or MANDATORY set membership. Regardless of the set membership modes, any changes in set membership must be explicitly given by DIMUI commands.

5.5.1 INSSET(rc,s,rp)

Purpose: Insert a member record occurrence into a set occurrence (similar to IDMS CONNECT command).

Parameters:

rc	-	(input) Stel for record occurrence to be inserted.
s	-	(input) Stel for set occurrence.
rp	-	(input) Stel for the occurrence of the preceding record in the set.

Usage: s is usually set up by a FINDST command (5.2.1). The user is responsible for maintaining the set order, i.e. he must access and specify the predecessor in the set. If the set insertion order is LAST or IMMATERIAL, the predecessor is the owner record; in this case, rp is the same as s.

Example. Suppose that rc points at an occurrence of an ORDER-ITEM record (possibly we just stored it by a STVIA or STORNX command), and we now want to CONNECT it to an occurrence of a PROD-ORD set. In the IDMS program we would now have to FIND a PRODUCT record. In DIMUI we issue a CALL FINDST(s1, 'PROD-ORD', 0,0,.FALSE.). If the set order in PROD-ORD is LAST (or if we somehow know that the ORDER-ITEM record will be the first in the set) we simply CALL INSERT(rc,s1,s1). Since we now probably no longer want the PRODUCT record, we free the stel by a CALL STEND(s1) command.

Now suppose above that the set PROD-ORD has a different order, i.e. the new record will have a predecessor other than its owner. In general this will be a random element of the set. Hence we follow the FINDST command by CALL FINDDN(rp,s1,0) (see 5.2.2). We now CONNECT the new record by CALL INSSET(rc,s1,rp). We will then free s1 and rp by STEND commands.

5.5.2 DISSET(r,s)

Purpose: Disconnect (remove) the specified record occurrence from the specified set occurrence (similar to the IDMS DISCONNECT verb).

Parameters:

r - stel for the specified record occurrence.
s - stel for the specified set occurrence.

Usage: The set is usually previously "established" by the ESTSET command. See 5.6.1 for an example of use.

5.6 Other stel commands

We list here several commands which have no IDMS DML counterparts.

5.6.1 ESTSET(r,s,SETNAM)

Purpose: Make the current occurrence of a record also the current of the indicated set.

Parameters:

r	- (input) stel for the current record occurrence.
s	- (output) stel for the set occurrence.
SETNAM	- (input) Name of the set (Hollerith).

Usage: In IDMS, whenever a record is brought into memory, it automatically becomes the "current of set" for each set of which it is a member or owner. DIMUI-IDMS, however, has no concept of "currency". In order to speak of a set occurrence, we must have a stel pointing to it. This is the function of the ESTSET command. It is used most commonly in conjunction with the FINDNX, FINDOW and DISSET commands

Example: Suppose that r1 is a stel which points at an occurrence of an ORDER-ITEM record. (Possibly we got to it by traversing the ITEM set for a given CUSTOMER-ORDER record.) We want to access its owner PRODUCT record. This is done by the two commands CALL ESTSET(r1,s1,'PROD-ORD') and CALL FINDOW(s1,r1). s1 now points at a PRODUCT record occurrence; r1 is unchanged. If we wanted to disconnect the order item from its PROD-ORD set, we would follow the ESTSET command by CALL DISSET(r1,s1).

Error-Stops:

95 - The current record is not of the type for members of SETNAM

96 - r is not a valid stel pointer, or there is an error in the "set bounds" descriptor (see 6.4).

26 - SETNAM is not in the schema.

5.6.2 DITTO(s,r)

Purpose: Copy stel s into a new stel r.

Parameters:

s - (input) pointer to a current stel.

r - (output) pointer to a new stel, set up to point to the same place as s.

Usage: This command is used whenever it is desired to move one stel through the data base while still keeping a pointer at the original place.

5.6.3 STEND(s)

Purpose: To free a stel when the element occurrence at which it points will no longer be required by the task.

Parameter:

s - (input) points at the element occurrence which is no longer required. On output, the value of s is undefined.

Usage: STEND should be used at the earliest opportunity when the task will no longer require the element occurrence. Otherwise, DIMUI risks running out of its internal storage, since there is a lot of information associated with each active stel.

5.7 Control Commands

5.7.1 CPUTIM(n)

Purpose: Increment processor time by n units, to simulate CPU-bound processing within the task.

Parameter:

n - (input) Positive integer, which specifies how many units of time are to be used.

Usage: CPU units are dimensionless. A processor description card supplied to Stage 3 (see 2.3E) translates these units into real time. Only a very gross estimate of CPU time is required, since DIMUI assumes that the computer is I/O-bound in any case. It is suggested that the user *not* use this command unless a significant amount of time (e.g. measured in milliseconds) is used by the task. For the PDP-11/70, one dimensionless unit = 1 millisecond. (See also below, Section 7.4.)

5.7.2 WRDISK(10,F)

Purpose: To increment the "work units" counter for the task by F.

Parameters:

10 - Must be written as above (identifies this as a "workunit" event).
F - Real (i.e. floating-point) number which is the number of units to add.

Usage: The primary use of this command is if a mix of tasks is being generated to be executed in a multiprogrammed multitask environment. Each task consists of a series of events; for example, a task which processes the addition of new order and order-item records, may consider the addition of one order record with its entire item set as an event. Another task, say one which processes the retrieval of all item records for a given product, may consider the retrieval of a single item record an event. If a number of different tasks are included in a benchmark of a particular system, e.g. if the simulator is run for a fixed period of simulated time, different weights would be assigned to the two different events. At the end of the simulation the throughput (weighted events per unit time) is computed; this gives a uniform measure of the processing capacity. (See Section 3.)

5.7.3 RANUN(DUMMY)

Purpose: A FUNCTION subprogram to generate a pseudorandom number between 0 and 1.

Parameters: No parameters are used, but since Fortran requires at least one parameter for a function, DUMMY may be used. The value returned is a floating-point number.

Usage: The most common use of this is in conjunction with control statements. Suppose that a certain condition causes the execution of a part of the task, which is skipped at other times. In the model we can't test the condition itself, but we do know that it is true (say) 25% of the time. The following sequence of commands models the true behaviour of the task:

```
      ....
      IF (RANUN(DUMMY) .GT.0.25) GOTO 1
      ....  (statements for part in question)
    1 CONTINUE
      ....
```

5.7.4 Common Random Numbers

In making comparative runs, it is sometimes important to keep random variability within the model to a minimum. For example, consider a task which accesses in sequence each member of a given set (or a random element of the set). Assume further that the number of members of the set may vary between great extremes. It is possible that a change in an unrelated parameter will cause different values to be generated for the number of members in the set (since the random-number sequence used within the model may be affected) thus causing a distortion of the true picture.

One way to overcome this problem is to make a number of different runs of the same task and organization (varying the starting random-number seed). The variations among the runs will cancel each other cut, and the average figure will (hopefully) reflect the true picture.

We have at our disposal another powerful technique for the reduction of the variance of the estimated difference. In principle we effect this technique by assuring that key variables in the system, which normally contribute a large part of the variability of the system operation, assume the same values at the two (or more) compared runs.

The method used in DIMUI for this purpose is to use the same stream of random numbers for a given key variable in all the compared runs. Thus the experimenter decides beforehand, through his knowledge of the database and the tasks, which instructions in the tasks are "strategic" in the sense of their contributing to the variation of the runs, and provides a sequence of random numbers for each such place. The instructions FINDST or SEARCH, for example, are often the subjects of this technique.

Since the random numbers in DIMUI are all generated by one global recursive generator, through an argument-less function, two short subroutines are provided to implement the procedure suggested above.

5.7.4.1 RANUNG(SEED)

Purpose: To obtain the current value of the random-number-generator seed.

Parameter:

SEED - (output) is an integer which contains the current seed.

Usage: When a task first starts executing, the program must initialize its internal storage for saving the seeds. For example, if there are four different "strategic" places where it is desired to keep separate seeds, the task may define an array SAVSED(4), and as part of the task initiation execute

```
     DO 10 I=1,4
10   CALL RANUNG(SAVSED(I)) .
```

Then, after executing the command at strategic point j, the command CALL RANUNG(SAVSED(j)) saves the next seed value for this particular series.

5.7.4.2 RANUNS(SEED)

Purpose: To set the random-number-generator seed to the next value of the desired series.

Parameter:

SEED - (input) is an integer which contains the seed-to-be.

Usage: See 5.7.4.1. Just before executing a command at strategic point j, the command CALL RANUNS(SAVSED(j)) resets the random-number-generator to generate the next number in this particular series.

5.7.4.3 Comment

This usage assures that a task is initiated in each of the compared runs with the same seed.

The location SAVSED(i) will contain during the run the successive values of the random-number stream used by the i-th "strategic" decision.

For more on the statistical methodology of using the model, see Reference [6].

6. HOW TO WRITE A DATA DESCRIPTION

Note This section is temporary, and will change entirely when the Stage 1 programs are completed.

6.1 Area Description (Type 1) cards

One for each area (see 2.1.A). These must be coded as follows:

Cols.	Contents (always right-justified in the field)
1-2	1 (identifies the type)
3-5	Index of the entry (1,2,...) This identifies the area number.
6-8	Index of the first extents entry for this area (see 6.2).
9-16	The page size for this area - 4.
17-19	For IDMS, must be 7. (For DIMUI in general, the OPEN routine to be used.)
20-22	Index of the buffer pool to be used for this area.
23-25	For IDMS, must be 7. (The Disk Space Management Routine to be used.)
26-33	Loading density of the area, specified as a percentage (integer $\leqslant$ 100). This is usually specified probabilistically (see 11).
34-35	For IDMS, must be 1. (Number of levels in a block-structured directory.)
36-39	If journal-tape entries are being written, index of the device which is the journal tape (see 7 3.3).
40-47	Should be blank.
48-52	Various flags, in hexadecimal. For IDMS, these must be coded as follows: X'1C0' if both "before" and "after" images are to be written to the journal tape; X'140' if only "before" images are to be written; X'180' if only "after" images are to be written; X'100' if no journal entries are to be written for this area.
53-55	Should be blank
56-80	Available for comments.

6.2 Area Extent Description cards (Type 2)

One or more for each area (see 2.3.A). These are to be coded as follows:

Cols.	Contents
1-2	2 (identifies the type)
3-5	Index of this entry (1,2,...) This identifies the particular extent.
6-8	Index of the next entry for this extent, or 0. All of the extents for one area form a linked list; the area description cards points to the first extent entry, that one points to the next, etc.
9-11	Index of the disk unit on which this extent resides.
12-15	Starting cylinder of this extent (cylinders are numbered from 0). For a fixed-head disk, should be specified as 0.
16-19	Starting track on each cylinder. (Tracks are numbered from 0. Note that an extent need not start at track 0. For example, an extent may occupy tracks 4-9 on a number of contiguous cylinders. A different area may be assigned a "parallel" extent on tracks 0-3 of the same cylinders.)
20-22	For IDMS, must be 1. (Starting block on the first track.)
23-29	Total number of pages in this extent.
30-33	Number of pages on each track of this extent. (The model does not compute this automatically on the basis of the known capacity of this device type.)
34-38	Number of tracks per each cylinder of this extent. If the device is a fixed-head disk, the total number of tracks in the extent should be specified here.
39-44	Page size plus disk storage overhead. Usually add 46 bytes.
45-80	Available for comments.

6.3 Element Description (Type 3) cards.

For every record type and every set type, there must be one Type 3 card. In addition, two such descriptors must be provided for each area. Before discussing the format of the Type 3 card, we show how the data structure of Figure 2 is presented to Stage 2. (Stage 1 will later generate such descriptions automatically.) This is shown in Figure 3.

First note that each element (area, set, and record) appears as a *node* of a tree. For ease of reference, we assign to each node an index (1,2,...) In our example, each record is the owner of at most one set; more generally, if a record owns several sets, the first will be linked to the second, the second to the third, etc. via the "next brother" link. In other words, we have a conventional "binary tree" representation of trees. Since a set consists of multiple occurrences of a record, that record points to itself as its brother.

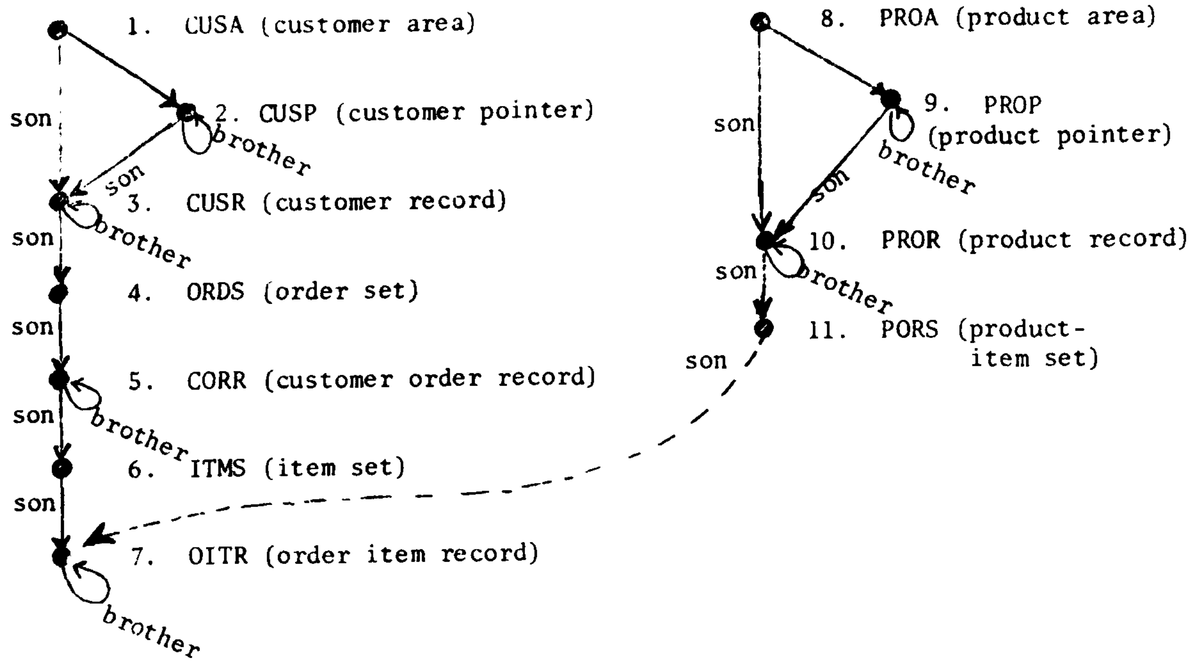
1. CUSA (customer area)
2. CUSP (customer pointer)
3. CUSR (customer record)
4. ORDS (order set)
5. CORR (customer order record)
6. ITMS (item set)
7. OITR (order item record)
8. PROA (product area)
9. PROP (product pointer)
10. PROR (product record)
11. PORS (product-item set)
son
brother

Figure 3 The schema of Figure 2 as input to Stage 2. Nodes 2,3,5,7,9 and 10 each indicates itself as its own brother. Nodes 1 and 8 never have a brother. Nodes 4,6 and 11 in our case happen to have no brothers, but in general if a record owns more than one set, the nodes are linked via the "next brother" field.

Second, we have introduced a new node between the area node and the CALC record in the area, which does not appear in Figure 2. This corresponds to a "start of CALC chain" pointer at the top of each page in IDMS. Conceptually, the CALC records of an area are divided into disjoint subsets; the records in one subset all hash to the same key and form a chain. This chain (which the user of IDMS does not see explicitly) corresponds to our new "pointer" node. In fact, it is represented in IDMS as a set ("system-owned"). This new node is however not a true son of the area node; the true son is the CALC record. The pointer node is an "auxiliary" son; this corresponds (roughly) in other systems to the notion of a directory used to allow direct access to an arbitrary record in a file. Both the area node and the pointer node point to the CALC record as a son.

We now describe the format of the Type 3 card.

Cols.	Contents
1-2	3 (identifies the type).
3-5	Index of this entry. This identifies the node.
6	Blank.
7-10	Unique name for this node. This is the name which is used in the task descriptions to refer to the element (see Section 5).
11-12	Blank.
13-15	Index of the son entry (or oldest son, if several). This is the "son link" field for representing the tree. 0 if no sons.
16-18	Index of the "next brother" entry; 0 if none. For records and "pointer" nodes, must point to itself. For an area, must be 0.
19-21	Index of the father node. For a CALC record, points to the area node. For an area node, this is 0.
22-24	For an area node, index of the "pointer" node ("auxiliary" son). For all other nodes, must be 0.
25-27	Index of the type 1 (area descriptor) card for this item.
28-31	"Records per Block". This parameter is significant only for sets whose members are stored VIA this set, and for a "pointer" node. Compute the record length plus the total size of all records stored via the sets which it owns (its sons). Since the number of sons may vary, use the expected value. If this total exceeds the page size, code 1 in this field. Otherwise code N, where N = pagesize/(total size of record plus its descendants), rounded to the nearest integer. For all other types of nodes, code 999.

Cols.	Contents (cont'd)
32-35	Total number of sons of this node. For an area node, total number of CALC records in this area. For a record, total number of sets which it owns. For a set, the number of members in this set (usually specified as a probability distribution).
36-40	Total record size, for a record. For a "pointer" node, code 8.
41-50	For an area, code 0. For a set whose members are stored VIA this set, or for a "pointer" node, expected total size of the descendant set, i.e. (expected number of members)* (record size + total expected size of all sets which it owns whose members are stored VIA this set). For a record, the sum of this attribute over all sets which it owns.
51-55	For IDMS, should contain 0. (In general, the size of key + pointer entries associated with pointer array.)
56-59	For IDMS, should contain 0. (In general, a maximum record + descendants size per page may be specified here.)
60-61	For IDMS, should contain 7. (Identifies the secondary-storage address-calculation routine.)
62-63	Code 1 for an area node, 0 otherwise. (In general, identifies this as a "directory root" node, and specifies the directory initialization routine.)
64-65	For IDMS, should contain 7. (Identifies what "special-action" routines to be used during insert operations.)
66-67	For IDMS, should contain 7. (Identifies the "special-action" routines to be used if a block-overflow condition is detected)
68-69	For IDMS, should contain 7. (Identifies the "available-space-in-block" calculation routine.)
70-73	Loading density for this file *in bytes*, i.e. how many bytes in a page are occupied. In general, this is loading percentage (see 6.1), multiplied by the page size. If the former is specified as a probability distribution, this parameter should also be specified as a probability distribution.
74-79	Flags, which identify various binary attributes. This field should be coded as follows for IDMS: 432 - This is an "area" node. 72 - This node is a "pointer" node. 30 - This node is a record. 102 - This node is a set. The following additional flags have to be added (in hexadecimal) to the above: 30 - Members are stored VIA this set; 4000 - This set has "prior" pointers; 8000 - This set has "owner" pointers.

6.4 Set bound (Type 11) Description Cards.

The descriptors described in this section are peculiar to the model and have no direct counterpart in IDMS. In general, if the members of a set are stored VIA the set, then all of the members of the set are "close" to each other in secondary storage. On the other hand, if the members are not stored via the set, there may still be a relationship in the storage location among the members of the set , and/or between the owner occurrence and the member occurrences. In the IDMS of course no additional descriptions are required, since exact pointers are stored in the data base together with the owner and member occurrences. The model, however, needs to generate such addresses, and requires additional information.

For example, let us assume (with reference to Figure 2) another set, called (say) "items of a given order over $1000 each", DEAR for short. The semantics of set membership for DEAR are such that for a given set occurrence all member occurrences are stored "near" to each other, the *bound* node being the ITEM set; but since the members are not stored via DEAR, the model would have no way of deducing this. The owner of this set may or may not be stored near the member occurrences. On the one hand, the owner may be the same ORDER record, in which case it is near. On the other hand, if we have another record type, say "overdue orders", which may own the set DEAR, there would not be any relationship in storage between this owner and the members, although the members of DEAR are near each other. In the first case, the *bound* between the owner and the members is the ITEM node; in the second, we say that there is no such bound.

The bound node may occur several levels above the member node. For example, assume another set (owned by ORDER records) called "supplements to this order in other orders by the same customer", whose members are ORDER-ITEM records. Note that in this case, what binds the members to one another and to the set owner is the fact that they all pertain to the same customer and are stored via the same ORDER set occurrence or its descendant ITEM set occurrences. The *bound* node is the CUSTOMER record.

This information is supplied to the model in Type 11 cards. The format of the card is:

Cols.	Contents
1-2	11 (identifies the type).
3-5	Index of the Type 3 entry for this set (identifies the set).
6-8	The bound node which binds the owner and member record occurrences (index of the corresponding Type 3 entry; 0 if no bound exists).
9-11	The bound which binds the member record occurrences to each other (index of the corresponding Type 3 entry; 0 if no bound exists).

7. HOW TO WRITE A "MEDIA DESCRIPTION"

Note: Extent descriptions were discussed in Section 6.2.

7.1 Buffer Pool Descriptions (Type 5) cards

These cards specify the buffer pools, one card per pool. A buffer pool may serve one or more areas. The format is as follows:

Cols.	Contents
1-2	5 (identifies the type)
3-5	Index of the entry (identifies the buffer pool)
6-8	For IDMS, should be set to 5. (Identifies the allocation strategy to be used.)
9-11	For IDMS, should be set to 1. (Identifies the buffer-release strategy to be used.)
12-14	Total number of buffers in the pool.
15-16	For IDMS, should be set to F. (Indicates that the buffers are to be shared among all active tasks.)
17-80	Available for comments.

7.2 Channel Descriptions (Type 6) cards

One of these cards must be provided for each channel in the computer system that is connected to a disk or tape unit. The format:

Cols.	Contents
1-2	6 (identifies the type)
3-5	Index of the entry (identifies the channel)
6-8	Index of the first unit connected to this channel
9-11	Index of the second unit connected to this channel (or blank)
...	
39-41	Index of the 12-th unit connected to this channel (or blank)
42-80	Available for comments.

Note: The order in which the units are listed specifies their servicing priority. Thus, if both units 1 and 2 are waiting for service and the channel becomes available, unit 1 is always served before unit 2, etc.

7.3 Unit Descriptions (type 7) cards

Each disk and tape unit in the simulated system must be described to the model by a Type 7 card. The format:

Cols.	Contents
1-2	7 (identifies the type)
3-5	Index of the entry (identifies the unit)
6-8	The device type. A table of the various device types currently supported is given in Appendix A.
9-11	For IDMS, should be 2. (Specifies the device scheduling strategy to be used.)

7.4 Processor Description (type 10) cards

For IDMS, one such card must be provided. Its format:

Cols.	Contents
1-2	10 (identifies the type)
3-5	Index of the entry (1). Identifies the processor.
6-9	CPU factor. This translates the dimensionless processor units (see CPUTIM command, 5.7.1) into microseconds. For the PDP-11/70, should be set to 1000.
10-13	Must contain 7E.
14-23	Must contain 100.
24-35	Must contain F.

8. HOW TO WRITE A JOB LOAD DESCRIPTION

In DIMUI, a multiprogrammed computer is viewed as consisting of n *work sources*, where n is the degree of multiprogramming. (The current system is generated for a maximum degree of multiprogramming = 4, but this can be increased by recompiling the system.) For each work source, the user must define a *work load*. A work load defines a succession of jobs to be run. A work load description is a sequence of cards ("type 8"), each of which identifies a job to be run, how much (if any) time to delay since the completion of the last job before starting this one, the execution priority of the current job, how many times in succession this job is to be run (with the same delay and the same priority each time) before starting the next one, and which job is to be run next.

Note in particular that the delay and the successor job may each be specified as probability distributions. This is very useful in simulating a multiterminal environment, where both the user's "think time" and the action which he may trigger can be characterized simply in the above terms.

When a job is translated by the Stage 2 modeller, it is assigned an identity by the user (which may be an integer between 1 and 4451). This job is saved in a special file called the EVF (see Section 11). If a multijob environment is being simulated, each of the jobs will have been assigned a unique number by the user and saved in the EVF by the Stage 2 modeller. (Remember that one task description can be used to generate many jobs; see Section 5.) These job numbers are specified in the work load description.

8.1 <u>The Work Load Description (Type 8) cards</u>

The format of the Type 8 cards is as follows:

Cols.	Contents
1-2	8 (identifies the type)
3-5	Index of the entry (1,2,...) Identifies this particular descriptor.

Cols.	Contents
6-15	Time (in microseconds) which must elapse before the job can be scheduled for execution. This models the user "think time", and is usually either 0 or specified as a probability distribution.
16-19	Identity of the job to be executed.
20-23	Execution priority of the job (1-9); higher = more urgent
24-27	The number of times this descriptor is to be "executed" repeatedly, before going to the next descriptor. Each time the current descriptor is executed, the delay time, job identity, and execution priority (all of which may be specified probabilistically) is computed again.
28-31	Index of the next descriptor be be "executed" after completing this one. There are two special values which may be coded here: 0 means that this work source is to be quiesced, i.e. no more work will be done here (if all work sources are quiesced, the simulation terminates); -999 means that the simulation is to stop as soon as this descriptor is completed. This is often used if it is desired to time one job in the presence of interference. The other work sources are defined as constantly working on some job mix, while the job in question is defined by a work descriptor with this code.

So far, we have not made any correspondence between a particular work source and a succession of descriptors. Note, however, that all we need to do is define a *starting* work descriptor for each work source; once a descriptor starts "executing", it specifies the next descriptor if any to be invoked. This representation gives us an economy in work load specification, as very few descriptors (often a single one) describe the load at several terminals.

Example 1. Suppose that we want to simulate a system with four terminals, which are all functionally similar, i.e. a user may invoke at any one of them any one of a number of jobs. Assume that the user behavior (i.e. his think time and the job he selects) do not depend on the terminal which he is using. Then the following one descriptor card needs to be supplied:

```
    8  1        -1  -2  1  1  1
```

The meaning of the parameters is:

The time delay is specified as -1. This indicates a probability distribution (see Section 9), which reflects the user "think time" behavior.

The job identity is specified as -2. This indicates a probability distribution which reflects the different jobs which he may select together with their respective probabilities.

The execution priority for all the possible jobs is specified as 1.

The repetition factor is specified as 1, i.e. it will be executed once before the successor descriptor is executed.

The "next descriptor" is indicated as 1, i.e. the same descriptor will be executed again (in an infinite loop). Each time it is executed, a new job and delay are generated.

Each of the work sources is given this as its starting descriptor. Since the work load as defined is infinite, the simulation will terminate after a fixed period of simulated time. Here the relevant output of the modelling run is probably the system throughput, and the queueing information at the various devices.

Example 2. Suppose that we want to time a job (say, 7) in the presence of interference, where the interference is the same job load as in Example 1, this time running on three terminals. We would add the following descriptor card:

```
8 2      0 7 1 1-999 .
```

We have specified the delay as 0, the job identity as 7, the repetition factor as 1, and the successor as -999 (end the simulation).

We now define the starting descriptor of work source 1 as 2, while the work sources 2,3, and 4 are assigned the starting descriptor 1. Here the simulation should terminate upon completion of work source 1, so we indicate the "ending simulated time" as infinite.

8.2 The NWO Card

The number of work sources (degree of multiprogramming) is indicated to the system by an NWO card. Its format is as follows:

Cols.	Contents
1-2	The degree of multiprogramming n.
3-5	For IDMS, should be 7. (Identity of the "system open" routine.)
6-8	For IDMS, should be 7. (Identity of the "system close" routine.)
9-80	Available for comments.

8.3 The SWD Card

The starting work descriptor for each work source is defined by a SWD card. Its format:

Cols.	Contents
1-3	Index of starting work descriptor (type 8 card) for first work source;
4-6	Index of starting work descriptor for second work source;
...	
3(n-1)+1 to 3n	Index of starting work descriptor for n-th work source.

9. HOW TO SPECIFY VALUES PROBABILISTICALLY

Many of the values which need to be specified in the descriptors (file loading density, number of elements in a set, any of the parameters in a "type 8" card, etc.) can be specified either as constants or as probability distributions.

If a value is specified as a constant, the constant appears in the appropriate field. If it desired to specify it probabilistically, the particular distribution must be defined and specified. It is specified by coding -n (n=1,2,...) in the appropriate field, where n identifies the particular distribution table.

Distribution tables are defined using "type T" cards. Their format are as follows:

New distribution table card:

Col. 1	Col. 3	Col. 4-9	Col. 11-80
T	T	n	Distribution information (described shortly)

Continuation card:

T			Distribution information

Comment card:

T	C		Comment (reproduced with current distribution on output)

The "distribution information" is free format, and may be continued on as many following continuation cards as desired. The comments card is used only to cause identifying information to be printed with Stage 1 output.

The following types of distributions are currently supported: *continuous*, *discrete*, *bilinear*, *collision*, and *"conditioned collision"*. Their meaning and formats are now explained.

9.1 Continuous Distribution

The user specifies pairs of numbers (x_i, y_i) $i = 0,1,\ldots N$, where $0 = x_o < x_1 < x_2 \ldots < x_N = 1$. The x_i's are the cumulative probabilities, and y_i are the corresponding values of the stochastic variable. That is, the variable y has the value $y_{i-1} \leqslant y \leqslant y_i$ with probability $(x_i - x_{i-1})$. Within this range, the values are distributed uniformly. The x_i and y_i values are specified as floating-point numbers with two places to the right of the decimal point (F8.2 format).

This distribution is specified by coding CON followed by as many pairs (x_i, y_i) (with parentheses, separated by blanks) as required.

Note: Because of the pecularities of the model, the values y_i for $i \geqslant 1$ must be specified as *one greater* than their true value.

Optionally, the user may specify the expected value of the distribution. This is used by the system only to check the specified distribution for possible errors: if the difference between the specified and computed expected values exceeds 10%, a warning message is printed out. The expected value is specified by coding EXP = value following the last (x_i, y_i) pair.

9.2 Bilinear Distribution

This is a special case of a continuous distribution, where the user specifies only the minimum, maximum, and expected values of the function. The cumulative distribution function is approximated as linear between 0 and b and between b and 1 (with different slopes); b is chosen to yield the desired expected value. This is described by specifying BIL followed by MIN = value, MAX = value, EXP = value as the distribution information. As usual, *value* is specified as a floating-point number with two decimal places.

9.3 Discrete Distribution

This distribution is similar to the continuous in that the user specifies pairs of values (x_i, y_i). Here, however, the distribution is discrete: the stochastic variable y takes on the value y_i with probability $(x_i - x_{i-1})$. This distribution is specified by indicating DES followed by the (x_i, y_i) pairs. Again, EXP = value may be optionally specified (see 9.1).

9.4 "Collisions" Probability Distribution

This distribution is specified for the "total number of sons" of a CALC "pointer" node. The system automatically computes the probability distribution function for the number of records "hashing" into the same page. The user must specify the following parameters in the "distribution information":

PAGES = number of pages in the area;
RECORDS = number of CALC records in the area.

These are specified as integers. Optionally the user may also specify the accuracy required for the approximation: POINTS = specifies (as an integer) the number of points the system is to use in linearizing the distribution (default: POINTS = 7); EPSILON = specifies (as a floating-point number to three decimal places) the maximum error allowed in dropping the "tail" of the distribution (default: EPSILON = .005). This distribution is selected by specifying COL at the start of the "information".

9.5 "Conditioned Collisions" Distribution

In modelling retrieval actions, the task "knows" that the requested CALC record will be found. I.e. regardless of the *a priori* distribution of the CALC records among the pages, we need to specify the probability for the number of sons of a CALC "pointer" node *given that it contains at least one record*. This is specified just as in 9.4, but now CCOL must be coded at the start of the "distribution information" field.

Note. In the current DIMUI implementation, CCOL (or, equivalently, a constant non-zero integer) must be specified for any CALC set on which a FINDST command will be performed. COL may be specified only in conjunction with a SEARCH command. In other words, DIMUI allows adding an element to a (possibly) empty CALC chain, but the FINDST command is not set up to handle a condition where it tries to find a known-to-be-there record in an empty chain.

10. HOW TO RUN THE THREE STAGES

10.1 Stage 1

10.1.1 Compiling a Task Description A task description is an ordinary Fortran IV subroutine (see Section V), and can be compiled using the Fortran-G compiler on the IBM 370. The output object module must be directed to the model library data set by including the DD card.

```
//FORT.SYSLIN DD DSN=
```

(the current data set name at Ft. Belvoir is CSCSATA.REITER.OBJECT; at the Technion it is ZCSSREZ.CSCSATA.REITER.OBJECT).

10.1.2 Compiling a Data Description The current version of the Stage 1 data translator is only concerned with translating the probability distributions into an internal form. The rest of the input cards are presented to Stage 1 in the same order as they are needed to Stage 2 and are passed on untranslated.

The translator reads card images (LRECL=80) from a SYSIN data set, writes card images (which will be the input for Stage 2 and/or Stage 3) on the OUT data set (LRECL=80), and prints messages on the SYSPRINT data set.

The translator is called by //EXEC PGM=ZNGNR, having included a JOBLIB or STEPLIB DD card for the load module library data set (Ft. Belvoir: DSN=CSCSATA.REITER.GO; Technion: DSN=ZCSSREZ,CSCSATA.REITER.GO). The order of the cards in the SYSIN data set are as follows.

10.1.2.1 *Seed Card:* contains a starting seed for the random number generator. This can be any integer right justified in columns 3-12.

10.1.2.2 *Ending Time* card: specifies whether Stage 2 or Stage 3 are to follow, and (for Stage 3) the simulated ending time. This is coded as follows:

Col. 2: T for Stage 2, F for Stage 3.

Cols.3-12: Amount of real time to be simulated in microseconds, right-justified. For Stage 2 (and for Stage 3 if the simulation ends after executing a job or a batch of jobs; see 2.4) code 9999999999

<u>10.1.2.3 *Dump*</u> card: specifies diagnostic dumps to be taken at system initialization and in case of error. These are normally meaningless to the user, and this card should be coded with 0's in all even-numbered columns (2.4,...80).

<u>10.1.2.4 *Type T*</u> cards: See Section 9.

<u>10.1.2.5 *Type 1*</u> cards: See Section 6.1.

<u>10.1.2.6 *Type 2*</u> cards: See Section 6.2.

<u>Note</u>: Although these cards must be supplied for Stage 2 as well as for Stage 3, only the total number of pages in the area are significant for Stage 2. Thus, the binding between a given area and its specific location in secondary storage may be changed without need to execute Stage 2 again, provided that the *page size* and the *total number of pages* in the area are not changed.

<u>10.1.2.7 *Type 3*</u> cards: See Section 6.3. These may be omitted for Stage 3.

<u>10.1.2.8 *Type 4*</u> cards: See Section 7.1. These may be omitted for Stage 2.

<u>10.1.2.9 *Type 6*</u> cards: See Section 7.2. These may be omitted for Stage 2

<u>10.1.2.10 *Type 7*</u> cards: See Section 7 3. These may be omitted for Stage 2.

<u>10.1.2.11 *Type 8*</u> cards: See Section 8.1. These may be omitted for Stage 2.

<u>10.1.2.12 *Type 10*</u> card: See Section 7.4. This card is required for both Stage 2 and Stage 3. However, the processor speed (CPU factor) may be altered without need to execute Stage 2 again.

10.1.2.13 *Type 11* cards: See Section 6.4. These may be omitted for Stage 3.

10.1.2.14 *Type 0* card: Zeros in columns 2 and 5; columns 6-80 may contain comments. Used to signify the end of the "type" cards.

10.1.2.15 *The following two cards are included only for Stage 2:*

1) BEGTSK card, specifies which task (TASKn, 6 ≤ n ≤ 20) is to be modelled (see 5). This is coded as follows:
   Cols. 1-3: n (right-justified);
   Col. 8: any integer or blank;
   Col.14: 1.

2) Job identity (JTNO) card; this identifies the output of Stage 2 (see Section 8). The job load description which will later be provided for Stage 3 will refer to this output by its job number. This card is coded as follows:

Cols. 1-5:	Job number, between 1 and 445. If a new job directory is to be initialized (see Section 9) -1 must be coded (the system will assign job number 1 to the present job). If a new job is being added to an existing directory, 0 must be coded; the next available job number will be assigned.
Cols. 6-10:	The start of the free space in the EVF where the output is to be written (see Section 11), must be an integer between 2 and 200. If the job number is coded as -1, this parameter should be coded as 2.

10.1.2.16 *The following two cards are included only for Stage 3:*

1) The NWO card (see 8.2).

2) The SWD card (see 8.3).

10.1.3 The Output of Stage 1

The program ZNGNR described in the preceding section is used to preprocess the input to each run of Stage 3. Its only function currently is to replace the "Type T" cards describing the distributions by "Type 4" cards (not described here) required by Stage 2 and Stage 3, and to replace the n specification in the fields by the appropriate pointers to the distribution table. The rest of the input to Stage 1 is transferred as is to the output data set.

10.2 Stage 2

10.2.1 Link-editing Stage 2 into a load module

After compiling the TASKn subroutines (task descriptions) as described in 10.1.1, a link-edit of the entire Stage 2 must be performed. This is done by running the standard linkage editor SYS1.LINKLIB(IEWLF880) with the options

```
          'MAP,LET,NOTERM,OVLY,SIZE=(128K,56K)'
```

and the following data set assignments:

```
//MYLIB DD DSN= (object module library, e.g. CSCSATA.REITER.OBJECT; see 10.1.1)
//SYSLIN DD DSN= (member MODEL in object library; e.g. CSCSATA.REITER OBJECT
                                                                (MODEL))
//SYSUT1 DD (temporary working data set)
//SYSLIB DD DSN= SYS1.FORTLIB
//SYSLMOD DD DSN= (partitioned data set member which will hold the load module.)
//SYSPRINT DD SYSOUT= A
```

10.2.2 Executing the Stage 2 load module

Once the Stage 2 load module has been compiled, it can be used to generate one or more jobs for the job library EVF. The input to it is the output of Stage 1. The output of Stage 2 is a new job in the EVF library; a listing of the input and some statistics about its run are printed on the message output data set.

The following data sets need to be provided for Stage 2:

```
//FT05F001 DD DSN= (the OUT data set of ZNGNR)
//FT06F001 DD SYSOUT=A  (this is the message output data set)
//FT08F001 DD  (a temporary data set with the attributes LRECL=80,
                any convenient block size, and enough space to hold
                400 logical records).
//FT09F001 DD DSN= (eventfile EVF).  This is a Direct (BDAM) data set,
                   with  BLKSIZE=1780, and SPACE=(1780,200).  For
                   more on this data set, see Section 11.
//FT12F001 DD DUMMY  (this data set can be used to dump some system
                      statistics).
```

If a number of runs of Stage 2 for the same data organization are desired, it is usually not necessary to re-run Stage 1. The necessary changes in the input data set (FT05F001) usually consist of changing a single card (e.g. the BEGTSK card - 10.1.2.15, or the random-number-generator seed - 10.1.2.1) and can be done manually, or by using the TSO EDIT program.

10.3 Stage 3

The Stage 3 load module does not usually need to be changed, and can be executed as supplied. Its inputs are (1) the output of the Stage 1 pre-processor; (2) the output of one or more runs of Stage 2 on the EVF. It is executed as follows:

```
//JOBLIB DD DSN= (partitioned data set containing the load module, e.g.
     CSCSATA.REITER.SIMUL at Ft. Belvoir, or ZCSSREZ.CSCSATA.REITER.SIMUL
     at the Technion).
//EXEC PGM=SIMUL
//FT05F001 DD (the OUT data set of ZNGNR)
//FT06F001 DD SYSOUT=A (this data set will contain the execution statistics)
//FT08F001 DD  (temporary data set as in 10.2.2)
//FT09F001 DD DSN= (eventfile EVF, as in 10.2.2)
//FT12F001 DD DUMMY
```

11. THE EVENTFILE EVF

This section describes the structure of the main interface between Stages 2 and 3: the eventfile EVF which is the library of "jobs".

A "job" is a stream of instructions for a "virtual data machine" VDM. This stream is "executed" sequentially, starting with the first instruction of a job, and ending with the final instruction which is necessarily "end of job". While it is not necessary for a user to know exactly how the VDM "works", it is helpful, and may provide some insight into the relationship between the logical file structures he adapts and system performance. We describe the VDM and the techniques for displaying the EVF later.

11.1 Space Management in EVF

The user is responsible for assigning space for the output of Stage 2 in the EVF, and must understand the structure of the EVF space.

The EVF is a BDAM ("regional 1" in PL/1, "direct" in Fortran) file. It consists of 200 blocks, each of 1780 bytes(445 words). (The 200-block figure can be changed, but several modules need to be recompiled.) Block 1 is a *job directory;* blocks 2-200 hold the instruction streams for the jobs in the library.

When a user runs Stage 2, he specifies the job number of the job which he is creating. (When the file is being initialized, he runs job 1; thereafter, he may add a new job with the next-highest job number, or re-create an existing job.) At the same time, he must also specify in which block to start writing the output (the first job output will normally start in Block 2). When Stage 2 finishes, the system prints out on the message output data set the address of the last block used for this job. The next block may then be specified as the starting block for the next job, etc.

Should the user forget which jobs occupy which space, a special utility DUMP is provided which (optionally) prints out the space usage map; this is described below.

The EVF is a *binary* file, and cannot be intelligibly printed out using the LIST or EDIT utilities; the DUMP utility is designed to perform this task.

If a previously-created job is no longer required, its space may be re-used. However, if the new job takes more space than the old one, the system allocates the blocks immediately following, and thus may overwrite existing jobs. No space-compression utility is currently provided.

11.2 The VDM instructions

<u>Note</u>: A full understanding of the rest of Section 11 is not required for operating the model.

The VDM is a page-oriented machine: most commands take the page address as the operand. A page address has three components: *file* (=area), *type* (which has no significance in IDMS), and *sequence number* (=page number within the area). The other components of a VDM instruction are *operation code*, and CPU *time* (in dimensionless units) which is required prior to the execution of this operation code. Thus, each instruction takes five words, and hence there are 89 instructions in each EVF block.

The operation codes are as follows:

1 - End of job (the other operands are not significant).

2 and 3 - Fetch (reference) the indicated page. The distinction between 2 and 3 is not relevant for IDMS.

4 - Release the indicated page. Operationally, this command does not do anything in the IDMS model; however, there should be a 4 operation corresponding to each preceding 2 operation for the same page.

10 - Increment "work units" counter (see 5.7.2). The "file" parameter contains the number of units increment (it is a floating-point number and does not print properly). The other parameters are not significant.

14 - Modify the indicated page.

15 - Call the space management routine to find space in a new page. (Although this command is sometimes generated, more generally the "space management page" is explicitly fetched and/or modified.)

The journalizing commands are not explicitly represented in the EVF; they are invoked implicitly while modelling "modify" commands.

It sometimes happens that a number of successive commands are identical in all five parameters: this happens for example when a list (which all happens to be on the same page) is being traversed. To save EVF space, a shorthand notation is utilized. An operation code of 0 means: the following instruction is to be repeated k times. k is contained in the *time* field of the current instruction; the other three parameters are not significant.

11.3 The DUMP Utility

This utility is designed to display the contents of the EVF. There are three modes of operation: (1) displaying the stream for a particular job; (2) displaying the contents of the job directory and the space occupied by the corresponding jobs; (3) displaying the entire contents of the EVF.

DUMP scans each block which it fetches for validity, and does not print out the contents of any block which contains invalid operations.

The utility is invoked as follows: A //JOBLIB DD or //STEPLIB DD card must specify the load module library (DSN=CSCSATA.REITER.GO for Ft. Belvoir, ZCSSREZ.CSCSATA.REITER.GO for the Technion). Then // EXEC PGM=GODUMP. The following DD cards must be provided:

```
//FT05F001 DD  (input data set for control)
//FT06F001 DD SYSOUT=A  (printed output data set)
//FT09F001 DD DSN= (eventfile EVF data set.
```

The input control data set contains one card. In columns 1-5 (right-justified) it must contain an integer n, with the following meaning:

n ≥ 2: Print out the instruction stream for a job whose first block in the EVF is n (this is mode (1) above).

n = 0: Display the contents of the job directory and the space occupied by the jobs.

n = -1: Display the contents of the entire EVF. Only valid blocks are printed.

12. CONTINUATION OF THE EXAMPLE

In this section we develop further the application of the model to the data base of Section 4. In the process we illustrate the use of the model in obtaining information on three separate tasks for the application.

12.1 Data Description

For the data structure of Figure 2 of Section 4 (see also Figure 3 of 6.3), let us make the following additional assumptions:

1) Each of the two AREAs CUSA and PROA has a page size of 1204 bytes;
2) CUSA has an allocated total of 2017 pages;
3) PROA has an allocated total of 14 pages;
4) The storage medium is a 2314-type disk, with 6 pages per track and 20 tracks per cylinder. CUSA was allocated space on cylinders 1-17 and PROA on cylinder 18 of the same disk unit.
5) There are 2000 CUSR (customer) records in CUSA, each 218 characters long.
6) There are a total of 1500 CORR (order) records. The distribution of these per one order-set occurrence ranges from 1 to 4 with an expected value of 1.5. Each CORR record is 230 characters long.
7) There are a total of 10,000 OITR (order item) records, each 90 characters long. The distribution of these per one item-set occurrence ranges from 1 to 5, with an expected value of 3. The distribution of OITR's per one product-item set occurrence ranges from 70 to 100, with an expected value of 90
8) There are a total of 100 PROR (product) records in PROA each 108 bytes long.
9) All sets are to be organized with "owner" and "prior" pointers.

The data description which is the input to Stage 1 is shown in Figure 4

Cards 1-16 describe the probability distributions which follow. These cards here appear at the beginning of the input, but may appear anywhere.

Cards 17-19 are respectively the starting seed for the random-number generator, the "ending-time" card, and the "dump" card.

Cards 20-21 are Type 1 cards, one for each area: note that the loading density is specified as probability distribution 1*, and the flags

* Distribution 1 says that the areas are 60-100 percent full, with an expected value of 80 percent.

```
[illegible]

              4      0         5      0     0 2 0 2            0
              1    230         5      0     0 2 1 2 2 2        0
              4      0        45      0     0 2 0 2            0
              0     90         0      0     0 2 0 2
              0      0         0      0     0 2 1 2 2 2        0
              1      0      1200      0     0 2 0   2 2
              1    108         0      0     0 2 0 2            0
              5      0         0      0         0 2   2
[illegible]
```

Figure 4. Input to Stage 1 for our example.

are set to "before and after images".

Cards 22-23 are the extend descriptions (Type 2) respectively for areas 1 and 2.

Cards 24-34 are the "Type 3" cards describing the structure of Figure 3. The "number of sons" for sets CUSP, ORDS, ITMS, PROP, and PORS is specified probabilistically. (Although the listing indicates the "node name" as five or six characters, only the first four are significant; the system ignores the contents of columns 11-12 of Type 3 cards.) The "total size" attribute (card columns 41-50) on cards 26-29 is also specified probabilistically; this could instead have been specified in terms of the (constant) expected value (see 6.3). The rest of the attributes are self-explanatory.

Card 35 is the processor description card.

Cards 36-38 are the "set bound" cards (see 6.4). They say in effect that members of sets ORDS (4) and ITMS (6) are stored VIA their respective owners, while for PORS (11) there is no relation in storage among set owners and/or members.

Card 39 is the "type 0" card, signifying the end of the "type" cards. Note that we have omitted cards of types 5-8; while they will need to be supplied for Stage 3, they are not needed for Stage 2.

Card 40 is the BEGTSK card, specifying that TASK10 is to be run (cols.1-5). This changes with the different runs of the Stage3; in our case, we also run TASK12 and TASK13.

Card 41 is the JTNO card. The -1 specifies that the EVF job directory is to be initialized to "one job". The 2 specifies that the EVF for this task is to be written starting with Record 2 (recall that record 1 is the directory; see 10.1.2.15 and 11).

Figure 5 shows the output of the Stage 1 program ZNGNR. Note that the probabilistic descriptions ("Type T" cards) have been replaced by "Type 4" cards, whose format is not described here, but which are functionally equivalent to the Type T cards. The probability pointers in the Types 1 and 3 cards have been changed accordingly. No other changes in the input have been effected. The output of ZNGNR is a sequential "card image" data set which becomes the input to Stage 2. In order to change some parameters for Stage 2 (e.g. the BEGTSK card or the JTNO card) it is not

```
        38345
 T         9999999999
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 1  1  1     1200   7   1   7         -1  1    1              100
 1  2  2     1200   7   1   7         -1  1    1              100
 2   1   0   1    1    0   1     2017    6    20    1250
 2   2   0   1   18    0   1       14    6    20    1250
 3   1 CUSAR    3   0   0   2   1 9991000     0          0     0     0 7 1 7 7 7    0    432
 3   2 CUSPTR   3   2   1   0   1    1   4    3       1200     0     0 7 0 7 7 7    0    [illegible]
 3   3 CUSREC   4   3   1   0   1  999   1  210         -7     0     0 7 0 7 7 7    0    30
 3   4 ORDSET   5   0   3   0   1    2 -11    0         -7     0     0 7 0 7 7 7    0  [illegible]
 3   5 CORREC   6   5   4   0   1  999   1  230        -15     0     0   1 7 7 7    0    [illegible]
 3   6 TIMSET   7   0   5   0   1   13 -19    0        -15     0     0 7 0 7 7 7    0  [illegible]
 3   7 DIFREC   0   7   6   0   1  999   0   90          0     0     0 7 0 7 7 7    0
 3   8 PROAR   10   0   6   9   2  999 100    0          0     0     0   1 7 7 7    0    [illegible]
 3   9 PROPTR  10   9   8   0   2   11 001    9       [illegible]     0     0 7 0 7 7 7    0
 3  10 PROREC  11  10   8   0   2  999   1  10[illegible]          0     0     0 7 0 7 7 7    0    [illegible]
 3  11 PORSET   7   0  10   0   2  999 -29    0          0     0     0 7 0 7 7 7    0  [illegible]
 4   1        2.00    80.00LOADING DENSITY OF ALL FILES
 4   2        0.00    60.00
 4   3        1.00   101.00
 4   4        2.00     1.50TOTSON  OF CUSPTR
 4   5        0.00     1.00
 4   6        1.00     3.00
 4   7        2.00   750.00SONSIZE OF CUSREC
 4   8        0.00   320.00
 4   9        0.82   751.00
 4  10        1.00  2721.00
 4  11        2.00     1.50TOTSON  OF  ORDSET
 4  12        0.00     1.00
 4  13        0.33     2.50
 4  14        1.00     5.00
 4  15        2.00   270.00SONSIZE OF CORREC & TIMSET
 4  16        0.00    90.00
 4  17        0.50   271.00
 4  18        1.00   451.00
 4  19        2.00     3.00TOTSON  OF  TIMSET
 4  20        0.00     1.00
 4  21        0.50     4.00
 4  22        1.00     6.00
 4  23        2.00     6.90TOTSON OF PROPTR
 4  24        0.00     1.00
 4  25        0.07     4.00
 4  26        0.42     7.00
 4  27        0.91    12.00
 4  28        1.00    17.00
 4  29        2.00    90.00TOTSON OF PORSET
 4  30        0.00    70.00
 4  31        0.33    91.00
 4  32        1.00   101.00
10   1 0.00   26          100 F
11   4   4   4
11   5   6   6
12  11   1   1
         END OF TYPE CARDS
        0      1
```

Figure 5. Output of the Stage 1 preprocessor ZNGNR. This is also the input for Stage 2.

necessary to return ZNGNR; its output can be changed directly, e.g. by using the TSO EDIT program.

Prior to describing the output of Stage 2 and the input to Stage 3, we will discuss task descriptions for three tasks for our application.

12.2 Task Descriptions for our Application

12.2.1 TASK10

For a random product, we need to get information on each of its customers. We do this by accessing the product record, traversing its product-order set PROR, and for each order-item (OITR) in it ascending first to its owner in the ITEM set (this gets us to the customer-order record CORR) and then to *its* owner in the ORDER set (this gets us to the customer record CUSR). To get an average timing figure, we do this for five different products.

The task description for this task is shown in Figure 6.

In line 140 a random occurrence of a product record is accessed, and the stel PORST is set up to point to the set occurrence PORS. In line 170 the stel OITST is set up to point to the ("0-th member" of the) set PORS, prior to traversing the set by FINDNX commands.

Line 200 moves the stel OITST to the next order-item record OITR, and line 210 tests for the end-of-set condition, which would signify that all items for this product have been traversed. Line 220 establishes the stel CORST to point to the current occurrence of the set ITMS (item set), and line 230 moves the stel CORST to the *owner* record, i.e. to the order record CORR. Lines 240 and 250 similarly set up the stel ORDST and move it to the customer record. Lines 260 and 270 erase the stels CORST and ORDST prior to moving to the next OITR in the set PORS.

After all OITR's have been traversed, the stels PORST and OITST must similarly be erased (lines 310 and 320). Note that the loop terminates by the GOTO from line 210 with OITST negative, therefore the minus sign in line 320.

```
      SUBROUTINE TASK10
      [illegible] PRODUCT, [illegible], ORDST, I, SEED
      [illegible] STARTING SEED
            [illegible] RANDNG(SEED)
C
C     REPEAT FOR [illegible] DIFFERENT PRODUCT RECORDS
C
      DO [illegible]
            CALL WDOLEN(10.[illegible])
      C [illegible] ALLOWS THIS OPERATION [illegible]
            RESTORE CURRENT SEED
            CALL RANDNG(SEED)
C     FIND PRODUCT RECORD
            CALL FINDST(PORST, [illegible], [illegible])
            SAVE CURRENT SEED
            CALL RANDNG(SEED)
            CALL [illegible](PORST, OLIST
C     [illegible] UNTIL END-OF-DATA IN PRODUCT [illegible]
                  CONTINUE
                  CALL [illegible](OLIST, PORST)
                  IF (OLIST .[illegible]. 0) GO TO [illegible]
                  CALL ESTSET(OLIST, CORST, 4H[illegible])
                  CALL FINDOW(CORST, OLIST)
                  CALL ESTSET(CORST, ORDST, 4HOR[illegible])
                  CALL FINDOW(ORDST, [illegible])
                  CALL [illegible]
                  CALL [illegible]
                  GO TO [illegible]
C
[illegible]
C
      [illegible]
      RETURN
      [illegible]
```

Figure 6. Listing of TASK10

Line 160 saves the current random-number seed; this is restored for the next product record. Thus this series of random numbers is independent of the processing in lines 200-250. If, for example, we were to change the set organizations to work without "owner" pointers, the FINDOW commands would generate a great many accesses to secondary storage and incidentally would also change the random number seed; the series for the next time through the loop would however not be affected.

Line 90 declares that one time through the loop is "worth" one work-unit; this is used in computing the system throughput in Stage 3.

12.2.2 TASK12

For a random customer, we need to know information about all products which he has on order. We do this by accessing the customer record, traversing the order set, and for each customer-order traversing the item set, accessing the owner product of the order-item record in the product-order set. To get an average timing figure, we do this for five different customers.

Figure 7 contains the task description for this task. Line 150 accesses a random customer record and positions the stel ORDST at an occurrence of the set ORDS. Then in line 180 the stel CORST is positioned at the set occurrence prior to traversing the order records.

Lines 220-230 access the next CORR record and test for the end-of-set condition. Lines 250 and 270 position the stel OITST at an occurrence of the ITEM set for the *current* CORR record, and set up the stel PORST to traverse the OITR records.

Lines 310-320 access the next OITR record and test for the end-of-set condition. Lines 340-360 successively set up the stel PROST to point to the current occurrence of the product-order set, access its owner, and then erase it.

Note that we use two different series of random numbers in this example. First, the seed is restored for each customer record. Second, the seed is also restored for each customer-order record in the order set. While this separation is not essential, it was put in when we were comparing the

```
              SUBROUTINE TASK12
      C
      C        MODEL OF A TASK WHICH RETRIEVES PRODUCT INFORMATION FOR EACH
      C        PRODUCT ON ORDER BY A GIVEN CUSTOMER. THIS TASK IS REPEATED
      C        FIVE TIMES.
      C
              INTEGER PORST,OITST,CORST,ORDST,I,SEED1,SEED2,PROST
              CALL RANUNG(SEED1)
              SEED2=SEED1
      C
              DO 60 I=1,5
      C
      C       FIND CUSTOMER RECORD
                    CALL RANUNS(SEED1)
                    CALL FINDST(ORDST,4HORDS 0,0,.FALSE.)
                    CALL WRDISK(10,1.)
                    CALL RANUNG(SEED1)
                    CALL DITTO(ORDST,CORST)
      C
      C----                ITERATE UNTIL END-OF-DATA IN ORDER SET
             20            CONTINUE
                           CALL FINDNX(CORST,ORDST)
                           IF (CORST .LE. 0) GO TO 50
                           CALL RANUNS(SEED2)
                           CALL FINDST(OITST,4HITMS,4HCORR,CORST,.FALSE.)
                           CALL RANUNS(SEED2)
                           CALL DITTO(OITST,PORST)
      C
      C----                      ITERATE UNTIL END-OF-DATA IN ITEM SET
             30                  CONTINUE
                                 CALL FINDNX(PORST,OITST)
                                 IF (PORST .LE. 0) GO TO 40
                                 CALL FSTSET(PORST,PROST,4HPORS)
                                 CALL FINDOW(PROST,PORST)
                                 CALL STEND(PROST)
                                 GO TO 30
             40            CONTINUE
                           CALL STEND(OITST)
                           CALL STEND(PORST)
                           GO TO 20
             50         CONTINUE
                        CALL STEND(CORST)
                        CALL STEND(ORDST)
      C
             60 CONTINUE
              RETURN
              END
END OF DATA
```

Figure 7. Listing of TASK12

the effects of different data organizations, e.g. the possibility of making the customer-order records CALC, or storing the order-item records VIA the product-order set. Since the number of member occurrences in a set was described probabilistically, it was important to keep the selected occurrence quantities the same across the various possibilities in question.

12.2.3 TASK13

We want to model the creation of the data base. We will assume that the creation process proceeds as follows. First, the file of product records is created. Then, we insert into the data base a customer record with all of his order records, and for each order record all of its items. This is repeated for each customer until the data base is full up to a certain density.

The building of the product file (which is in any case small) is not modelled here. We do want to model the rest of the creation. Rather than build the entire file, we "sample" the time it takes to insert a group of customer records at various loading densities, e.g. at 10, 30, 50, 70 percent of file capacity. Each sample consists of inserting five customer records together with their order and item sets.

Each "sampling" is done by the same task, shown in Figure 8. This task is run with the data shown in Figure 4, changing the BEGTSK card to 13, the CCOL's to COL's and changing the file loading density as desired. The explanation of the task is as follows.

Line 300 stores a new customer record. Lines 330 and 350 position the stel ORDS at the order set, and set up the stel CUSORD for successively storing customer-order records.

Lines 400 and 410 store the next customer-order record and test for end-of-set (i.e. the set being exhausted). Lines 430 and 450 position the stel ITEMS at the ITEM set and set up the stel ORITEM for successively storing order-item records.

Lines 500 and 510 store the next order-time record and test for end-of-set. Line 530 finds a random occurrence of a product record in order to

```
      SUBROUTINE TASK13
C
C     CREATION OF DATA BASE FOR IDMS EXAMPLE APPLICATION.
C     PRODUCT FILE IS ASSUMED TO EXIST. WE INSERT ONE
C     CUSTOMER RECORD, ITS ENTIRE ORDER SET, AND (FOR EACH
C     ORDER) ITS ENTIRE ORDER ITEM SET. THE ITEM RECORD IS
C     CONNECTED TO ITS PRODUCT ORDER SET. FIVE CUSTOMER RECORDS
C     ARE THUS INSERTED. THIS TASK IS RUN FOR SEVERAL DIFFERENT
C     LOADING DENSITIES ("SAMPLE" FOR THE CREATION PROCESS), AND
C     THE TIME TO CREATE THE ENTIRE DATA BASE IS EXTRAPOLATED.
C
      INTEGER CUSR,ORDS,CUSORD,ITEMS,ORITEM,PROORD,SAVSED(4)
C
C     CUSR   - STEL FOR CUSTOMER RECORD
C     ORDS   - STEL FOR ORDER SET
C     CUSORD - STEL FOR CUSTOMER-ORDER RECORD
C     ITEMS  - STEL FOR ITEM SET
C     ORITEM - STEL FOR ORDER-ITEM RECORD
C     PROORD - STEL FOR PRODUCT-ORDER SET
C     SAVSED - ARRAY OF CURRENT SEEDS FOR FOUR SERIES
C
```

Figure 8. Listing of TASK13 (1 of 2)

```
         DO 1 J=1,4
       1 CALL RANUNG(SAUSED(J))
   C
         DO 100 I=1,5
         CALL WRITE[illegible](I,[illegible])
   C
   C              ** STORE A NEW CUSTOMER RECORD **
                  CALL RANUNS(SAUSED(1))
                  CALL SEARCH(CUNR [illegible])
                  CALL RANUNG(SAUSED(1))
                  CALL RANUNS(SAUSED(2))
                  CALL FINDST(ORDS,AHORDS,AHCUSR,CUSR,.FALSE.)
                  CALL RANUNG(SAUSED(2))
                  CALL [illegible](ORDS,[illegible])
   C
                  WHILE (NOT END OF DATA [illegible]) DO
   C                   ** STORE [illegible] **
      10                       CONTINUE
                               CALL STORNX(CUSORD,ORDS)
                               IF (CUSORD .LT. 0) GOTO 50
                               CALL RANUNS(SAUSED(3))
                               CALL FINDST(ITEMS,ARTICS,AHORDS,CUSORD,[illegible])
                               CALL RANUNG(SAUSED(3))
                               CALL [illegible](ITEMS,ORITEM)
   C
                               WHILE (NOT END-OF-DATA IN ITEM SET) DO
   C                                ** STORE ITEM RECORD [illegible] **
      20                                      CONTINUE
                                              CALL STORNX(ORIT[illegible]ITEMS)
                                              IF (ORITEM .LT. 0) GOTO 30
                                              CALL RANUNS(SAUSED(4))
                                              CALL FINDST(ARTIC,AHPORS,0,0,[illegible])
                                              CALL RANUNG(SAUSED(4))
                                              CALL INSERT([illegible],PROORD,[illegible])
                                              CALL [illegible]
                                              GOTO 20
   C
   C
      30                       CONTINUE
                               CALL STENDS(ORITEM)
                               CALL STEND(ITEMS)
                               GOTO 10
   C
   C
      50             CONTINUE
                     CALL STENDS([illegible])
                     CALL STEND(ORDS)
                     CALL STEND(CUSR)
     100 CONTINUE
   C
         RETURN
   C
         END
```

Figure 8 Listing of TASK13 (2 of 2)

connect the newly-stored order-item record to its product-order set (in ORDER IS LAST sequence). The connection is done in line 550.

In this task, four different random-number series were use. .

12.3 Stage 2 for our example

The data shown in Figure 5 serves as the input data set (//FT05F001 DD ...) for a run of the program MODEL. It will be recalled that there are two sets of outputs from this run: the printed output (//FT06F001 DD SYSOUT=...) and the EVF which forms part of the input for Stage 3.

The printed output is shown in Figure 9. The input is listed first. After printing the JTNO= (which is the information from the JTNO card, except that the actual job number assigned is printed), the rest of the output results from execution of the program. The first message printed (LAST RECORD WRITTEN WAS ...) informs the user how much of the EVF was used. The rest of the output is meaningless for IDMS and may be ignored.

Figure 10 shows an initial segment obtained from dumping the EVF (see Section 11). Recall that the EVF is a binary file and cannot be listed directly; the output of Figure 10 was obtained using the DUMP utility (11 3).

12.4 Stage 3 for our example

In our example we are only interested in timing TASK10, without interference from other concurrent jobs and without multiprogramming. Figure 11 shows the input data set for SIMUL. This data set was produced by ZNGNR from input similar to Figure 4. Note, however, that Type 3 and 11 cards are omitted.

Card 12 is a buffer pool description card (7.1): it specifies that eight buffers are to be used for both of the areas. Note that the "area description" cards (4 and 5) each specify that buffer pool 1 is to be used, i.e. they share the buffer pool.

Card 13 is a channel description card. It specifies that the (one) channel connects to unit 1 (a disk) and unit 2 (a tape).

```
THE SEED IS         08345
MODEL=T END TIME=            599
INPUT LISTING FOLLOWS

 1  1  1      1200  7  1  7         -1 1   1                100
 1  2  2      1200  7  1  7         -1 1   1                100
 2  1  0  1   1   0  1   2017   6    20    1250
 2  2  0  1  10   0  1     14   6    20    1250
 3  1 USAR   3  [illegible] 0  2  1 9991000    0          0    0    0 7 1 7 7 7   0   [illegible]
 3  2 OUTPTR [illegible] [illegible] 1 [illegible] 1  1  -4    8       1200    0    3 7 [illegible] 7 7 7   [illegible]   [illegible]
 3  3 CUSPEC  4  [illegible] 1 [illegible] 1 999  1  [illegible]      -7    0    0 7 0 7 7 7   0   [illegible]
 3  4 CRSET   3  0  3  0  1  [illegible] -11   0          -7    0    0 7 0 7 7 7   0  0132
 3  5 CORREC  6  5  4  0  1 999   1  250        -15    0    0 7 1 7 7 7   0   [illegible]
 3  6 TIMSET  7  [illegible] 5 [illegible] 1  1[illegible] -1[illegible]  0        -1[illegible]    0    0 7 [illegible] 7 7 7   0  01[illegible]
 3  7 DITREC  9  7  6  0  1 999   0   50          0    0    0 7 9 7 7 7   0   [illegible]
 3  [illegible] RECAR 10 0  0  0  [illegible] 999 100   0          [illegible]    0    0 7 1 7 7 7   0   [illegible]
 3  9 PROPTR 10 9  8  0  2  11 -2[illegible]   8       1200    0    0 7 0 7 7 7   0   [illegible]
 3 10 PROREC 11 10 [illegible] 0  2 999   1  100          0    0    0 7 0 7 7 7   0   [illegible]
 3 11 PERSET  7  9 10  0  2 999 -2[illegible]  0          0    0    0 7 0 7 7 7   0  010[illegible]
 4  1      2.00      9.00LOADING DENSITY OF ALL FILES
 4  [illegible]      0.00     50.00
 4  [illegible]      1.00    101.00
 4  4      2.00      1.50TETSON OF CUSPTR
 4  [illegible]      0.00      1.00
 4  6      1.00      3.00
 4  7      2.00    75[illegible].00TSIZE OF CORREC
 4  [illegible]      0.[illegible]0    32[illegible].00
 4  [illegible]      [illegible]     751.00
 4 10      1.00   3721.00
 4 11      2.00      1.50TETSON OF CRDSET
 4 12      0.00      1.00
 4 1[illegible]      0.[illegible]      2.5[illegible]
 4 14      1.00      5.00
 4 1[illegible]      2.00    170.00TSIZE OF CORREC & TIMSET
 4 16      0.00     [illegible]
 4 17      0.50    271.00
 4 18      1.00    451.00
 4 1[illegible]      2.00      3.00TETSON OF TIMSET
 4 20      0.00      1.00
 4 21      0.50      4.00
 4 22      1.00      5.00
 4 23      2.00      6.00TETSON OF PROPTR
 4 24      0.00      1.00
 4 25      0.07      4.00
 4 26      0.42      7.00
 4 27      0.95     12.00
 4 28      1.00     17.00
 4 2[illegible]      2.00     90.00TETSON OF PERSET
 4 30      0.00     70.00
 4 31      0.23     91.00
 4 32      1.00    101.00
10  11000   7E          100 F
11  4  4  4
11  6  6  6
11  11  1  1
 9  [illegible]    END OF TYPE CARDS

END OF INPUT DATA.
```

Figure 9 Output of Stage 2 (1 of 2)

```
END OF TABLES DUMP.
BEGINNING TASK ISIO LAST LOGICAL RECORD IS     9 EVENT WEIGHT=      1
JTNO=     1 #VALR=     2
LAST RECORD WRITTEN WAS    39



 SUMMARY TOTALS

 BLOCKS BY BLOCKTYPE
 TYPE   TOTAL
          14
         942

 FILE LEVELS, DESCEX(GENERATED)-STOTIL(COMPUTED)

BETWEEN   AND      TOTAL
 -32767   -2000        0
  -1999   -1000        0
   -999    -500        0
   -499    -300        0
   -299    -200        0
   -199    -150        0
   -149    -100        0
    -99     -50        0
    -49     -20        0
    -19       0     2758
      1      20     3739
     21      50        2
     51     100      942
    101     150        0
    151     200        0
    201     300       62
    301     500        0
    501    1000        2
   1001    2000        0
   2001   32767        3


 *** MISCELLANEOUS AREA DUMP ***
INPUT UNIT =    5 (INUNIT)
OUTPUT UNIT =    6 (OTUNIT)
NULL POINTER =          0 (NULL)
MODELLING = T (MODEL)
MASTER CLOCK  (USEC) =           0 (MSTCLK)
```

Figure 9 Output of Stage 2 (2 of 2)

```
JOB NUMBER          2


RECORD          2
OPCODE      TIME   [illegible]  [illegible]  [illegible]
     2        0        2                   [illegible]
     0        0        2                        5
     3        1    [illegible]
     [illegible] 00000000           00000000
     2        0        2                        5
     3        0        2                        5
     [illegible] 0     1                 [illegible]
     3        0        1                 [illegible]
     3        0        1                  [illegible]
     [illegible] 0     1                 [illegible]
     [illegible]
     3   [illegible]   1                 [illegible]
     [illegible]
     2   [illegible]   1                 [illegible]
     3   [illegible]
     3        0        1                 [illegible]
     4        0        1                 [illegible]
     4        0        1                 [illegible]
     3   [illegible]   1                 [illegible]
     4        0        1                 [illegible]
     2        0        1                 [illegible]
     3   [illegible]   1                 [illegible]
     3        0        1                 [illegible]
     4        0        1                 [illegible]
     [illegible] 0     1                 [illegible]
     [illegible] 0     1                 [illegible]
     3   [illegible]   1                 [illegible]
     4        0        1                 [illegible]
     3        0        1                 [illegible]
     3        0        1                 [illegible]
     4        0        1                 [illegible]
     4        0        1                 [illegible]
     3        0        1                 [illegible]
     4        0        1                 [illegible]
     [illegible] 0     1                 [illegible]
     3        0        1                 [illegible]
     3        0        1                 [illegible]
     4        0        1                 [illegible]
     [illegible] 0     1                 [illegible]
     3        0        1                 [illegible]
     4        0        1                 [illegible]
     2        0        1                      713
     2        0        1                 [illegible]
     4        0        1                     1011
     4        0        1                     1013
     4        0        1                     1011
     2        0        1                      713
     4        0        1                      713
     [illegible] 0     1                     1999
     3        0        1                     2003
     3        0        1                     2001
     4        0        1                     2003
     4        0        1                     2001
```

Figure 10. EVF Dump

```
[illegible]
```

Figure 11. Input to Stage 3

Card 14 is a unit description card. The device type is specified as 5, which is a 2314-type disk.

Card 15 is another unit description card. The device type is specified as 10, which is a tape.

Card 16 is a work load description card. In our case, only one job is to be executed, and this is job 1. The parameters specify a delay of 0 (the job is to be scheduled immediately), a priority of 1, a repetition factor of 1 (the job is executed one time only), and a "next descriptor" of -999, i.e. the simulation is to end when this job terminates.

Card 19 specifies the degree of multiprogramming as 1 (NWO card).

Card 20 specifies the starting work descriptor for the first (and only) work source as 1, i.e. it points to card 16 (SWD card).

The output of the Stage 3, and the final output of DIMUI for our example, is given in Figure 12. First, a recap of the input is given. Then appears the line MSTCLK = ... This is the reading of the Master clock at the end of the simulation (in microseconds); in our case it says that the job took 70.8 seconds of simulated time to execute. The task in question was TASK10 (see 12.2.1); that is, it takes an average of 14.2 (70.8/5) seconds to process the customers of a given product. This is the main output of SIMUL: what follows are some details of component utilization.

The next line says that the (one) processor (CPU) was active 22.6 seconds, which is 31.89 percent of the time.

The "log file" is not being used in this example.

Next follows information on unit utilization Unit 1 (the disk) was used 48.2 seconds, 68.1 percent of the time. It spent 0 time waiting for a channel (in our example there is no contention for the channel). Of the total disk accesses which were made, 516 were on the cylinder on which the arm was already positioned, i.e. there were no seeks associated with these requests.

The rest of the units (12 are printed out) were not used at all.

Next follows a matrix of information on queue sizes at the various units. The leftmost column identifies the unit, while successive columns

```
THE SEED IS          12345
MODELED END TIME=    999000000
INPUT LISTING FOLLOWS

 1  1  1     1200  7  1  7        -1 1   2              100
 1  2  2     1200  7  1  7        -1 1   2              100
 2  1  0  1   1   0  1   2017   6   20  1250
 2  2  0  1  18   0  1     14   6   20  1250
 4  1     2.00   30.00  LOADING DENSITY FOR COPAR + PROAK-8[illegible]
 4  2     0.00    0.00
 4  3     0.20   81.00
 4  4     1.00  101.00   ***
 5  1  5  1  3 F        EIGHT PUBLIC BUFFERS
 6  1  1  2
 7  1  5  2             'LINEAR' 2314-A DISK,SCAN SCHEDULING
 7  2 10  1
 8  1          0   1   1   1-999
10  11000  7E        100 F
 0  0   END OF TYPE CARDS

END OF INPUT DATA.


END OF TABLES DUMP.
THERE ARE  1 WORK SOURCES. SYSTEM OPEN ROUTINE IS  7. SYSTEM CLOSE ROUTINE IS  7
THE INITIAL JOBS ARE   1
MSTCLK=     70836112



PROC       BUSY        %        IDLE       %
   1  22592000  31.89  48244112  68.11
      1 RECORDS WRITTEN ONTO LOG FILE.

UNIT SPACE (USEC) % OF MSTCLK   WAIT FOR CHANNEL % OF MSTCLK # OF ZERO SEEKS
   1     48245064       68.11                   0       0.0             516
   2            0        0.0                    0       0.0               0
   3            0        0.0                    0       0.0               0
   4            0        0.0                    0       0.0               0
   5            0        0.0                    0       0.0               0
   6            0        0.0                    0       0.0               0
   7            0        0.0                    0       0.0               0
   8            0        0.0                    0       0.0               0
   9            0        0.0                    0       0.0               0
  10            0        0.0                    0       0.0               0
  11            0        0.0                    0       0.0               0
  12            0        0.0                    0       0.0               0
```

Figure 12   Output of Stage 3 (1 of 2)

```
UCB                                            QSIZE
           0           1           2           3           4           5           6

  1   22592000    48244112           0           0           0           0           0
  1    31.89 %     68.11 %     0.0 %       0.0 %       0.0 %       0.0 %       0.0 %

  2   70836112           0           0           0           0           0           0
  2   100.00 %     0.0 %       0.0 %       0.0 %       0.0 %       0.0 %       0.0 %

  3   70836112           0           0           0           0           0           0
  3   100.00 %     0.0 %       0.0 %       0.0 %       0.0 %       0.0 %       0.0 %

  4   70836112           0           0           0           0           0           0
  4   100.00 %     0.0 %       0.0 %       0.0 %       0.0 %       0.0 %       0.0 %

  5   70836112           0           0           0           0           0           0
  5   100.00 %     0.0 %       0.0 %       0.0 %       0.0 %       0.0 %       0.0 %

  6   70836112           0           0           0           0           0           0
  6   100.00 %     0.0 %       0.0 %       0.0 %       0.0 %       0.0 %       0.0 %

  7   70836112           0           0           0           0           0           0
  7   100.00 %     0.0 %       0.0 %       0.0 %       0.0 %       0.0 %       0.0 %

  8   70836112           0           0           0           0           0           0
  8   100.00 %     0.0 %       0.0 %       0.0 %       0.0 %       0.0 %       0.0 %

  9   70836112           0           0           0           0           0           0
  9   100.00 %     0.0 %       0.0 %       0.0 %       0.0 %       0.0 %       0.0 %

 10   70836112           0           0           0           0           0           0
 10   100.00 %     0.0 %       0.0 %       0.0 %       0.0 %       0.0 %       0.0 %

 11   70836112           0           0           0           0           0           0
 11   100.00 %     0.0 %       0.0 %       0.0 %       0.0 %       0.0 %       0.0 %

 12   70836112           0           0           0           0           0           0
 12   100.00 %     0.0 %       0.0 %       0.0 %       0.0 %       0.0 %       0.0 %

CHANNEL USAGE (USEC) % OF SSTCLK
    1      20394320       28.79

I/O SUMMARY. BLOCKS TRANSFERRED BY TCB.
 NREC( 1)      0  NREC( 2)  1412  NREC( 3)      0  NREC( 4)      0  NREC( 5)      0

EVENT COUNT
 EVT( 1)=   0.0  EVT( 2)=    5.0  EVT( 3)=    0.0  EVT( 4)=    0.0  EVT( 5)=    0.0

TOTAL EVENTS=     5.0 THROUGHPUT=    0.007 RECORDS/SEC
```

Figure 12 Output of Stage 3 (2 of 2)

contain the time (absolute and percentage) that the device queue length was 0, 1, 2 etc. requests. In our case, since there is no multiprogramming (and no overlapping of requests), no device shows a queue length of more than 1. Zero length denotes idle time. Thus this information in our case merely repeats the previous.

Next follows information on the (one) channel usage: it is shown to have busy 20.4 seconds, or 28.8 percent of the time.

The next two lines list the total number of I/O requests by the various tasks (1412 disk requests by our one task).

The next two lines indicate how many work units (mislabeled "events" in the output) were processed by the various tasks; it shows that 5 were processed by our one task.

Finally, the system throughput ("events" per unit time) is computed.

APPENDIX A. "Device Type Table" Entries

The *device type table* describes the salient characteristics of each device type (disk and tape drive) connected to the system. This table is not supplied by the user, but is assembled into the system. This appendix describes the various devices currently available. Information on how to add new device types to the system is not provided here.

Index	Description of the device
1	IBM 2314 Model A disk. 25 milliseconds per revolution, 7403 bytes per track, no RPS (rotational position sensing).
2	IBM 2314 Model 1 disk. Similar to Model A, but somewhat different 'seek' characteristics.
3	IBM 3330 Model I disk. 16.7 milliseconds per revolution, 13030 bytes per track, RPS, 103 bytes per sector.
4	IBM 3330 Model II disk. Similar to Model I, but different 'seek' characteristics.
12	DEC TJU-16 tape drive. Tape speed at full velocity = 7200 chars/second, interrecord gap = 800 characters, time required to come to a complete stop from full speed = 5 milliseconds, tape rewind speed = 240000 chars/second.

R E F E R E N C E S

1. Reiter, A., "On Performance Modelling of Data Base Management Systems — An Inductive Approach". MRC Technical Summary Report 1648, Mathematics Research Center, University of Wisconsin, July 1976, 42 p.

2. Reiter, A., and Finkel, B., "Simulating a Virtual Data Machine". MCR Technical Summary Report 1626, Mathematics Researh Center, University of Wisconsin, May 1976, 53 p.

3. Reiter, A., "Data Models for Secondary Storage Representations". Proceedings, Very Large Data Bases Conference, Sept. 1975, ACM, pp. 87-119.

4. Reiter, A. and Sibley, E., "Simulation and Data Administration". To appear.

5. Cullinane Corp., IDMS DDL and DML reference guides. Cullinane Corp., Wellesley, Mass., April 1975.

6. Hofri, M., and Reiter, A., On Statistical Methodology in Data Base Simulation (in preparation).

ATE

LMED

-8